RADC-TR-88-129
Final Technical Report
July 1988

A SURVEY OF PARALLEL COMPUTING

Amherst Systems, Inc.

Susan E. Miller

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss AFB, NY 13441-5700

A SURVEY OF PARALLEL COMPUTING

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-129
6a. NAME OF PERFORMING ORGANIZATION Amherst Systems, Inc. — 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COEE)
6c. ADDRESS (City, State, and ZIP Code) 30 Wilson Road Buffalo NY 14221	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center — 8b. OFFICE SYMBOL (If applicable) COEE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-87-C-0082
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO.: 65502F — PROJECT NO.: 3005 — TASK NO: RA — WORK UNIT ACCESSION NO.: 77
11. TITLE (Include Security Classification) A SURVEY OF PARALLEL COMPUTING	
12. PERSONAL AUTHOR(S) Susan E. Miller	
13a. TYPE OF REPORT Final — 13b. TIME COVERED FROM Jun 87 TO Feb 88	14. DATE OF REPORT (Year, Month, Day) July 1988 — 15. PAGE COUNT 200
16. SUPPLEMENTARY NOTATION N/A	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Non-von Neumann Architectures; Software Tools; Parallel Computing; Software Life Cycle; Interconnection Networks

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report summarizes the research accomplished under a Small Business Innovation Research (SBIR) Phase I contract F30602-87-C-0082. Until recently, software engineering technology has focused on the use of conventional von Neumann computer architectures. Current computer systems feature continually increasing computational resource requirements which have become unfeasible or unrealizable using conventional von Neumann techniques. Non-von Neumann architectures featuring multiple processing elements offer solutions that achieve the application requirement. However, software engineering technology remains oriented towards von Neumann methodologies. This, coupled with the applications specific nature of non-von Neumann architectures, has resulted in a fragmented software engineering environment.

This effort provided a very thorough view of many of the software engineering problems that are currently associated with the utilization of non-von Neumann computers. The results of this study support current and future applications programs by providing gound rules to architecture selection and system and software life cycle definition. (over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Paul M. Engelhart	22b. TELEPHONE (Include Area Code) (315) 330-4476 — 22c. OFFICE SYMBOL RADC (COEE)

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Block 19. Abstract (Cont'd)

Specifically, this report provides a very comprehensive and unique description of the non-von Neumann terminology, architectures, machines, languages, and tools prevalent in industry and academia. It assumes of the reader only basic von Neumann literacy and continues into considerable detail.

CONTENTS

PREFACE

This document represents the findings of a six month long research effort (July, 1987 to January, 1988) to survey software features of non-von Neumann architectures, including parallel computers, systolic arrays, and dataflow machines. Rome Air Development Center (RADC), located at the Griffiss Air Force Base in Rome, New York, provided Amherst Systems Inc., a private research firm located in Buffalo, New York, with the funds to perform the research (contract F30602-87-C-0082).

This document serves to familiarize the reader with the underlying concepts of parallel computing. Chapter 1 defines basic parallel computing terminology and concepts in order to prepare the reader for the particular terminology which is adopted throughout this document. Chapter 2 surveys the various ways to link processors to processors and processors to memories in parallel computers (i.e., interconnection networks). Chapter 3 surveys various commercially available and research parallel computers. Chapter 4 discusses high level programming languages for parallel computers and describes several existing and proposed languages. Chapter 5 discusses the various programming techniques used for communication and synchronization between processors in a parallel computer. Chapter 6 discusses various issues associated with designing algorithms for parallel computers and gives a few illustrated examples. Chapter 7 discusses operating systems issues for parallel computers. Chapter 8 surveys various software tools available for parallel computers. Chapter 9 provides an extension to the conventional software life cycle (of serial computers) for parallel computers. Chapter 10 discusses various supercomputer research centers, most of which are found in the United States. Chapter 11 discusses research topics which

need to be addressed by the scientific community. Chapter 12 gives a list of journals and books in which the reader will find more information on parallel computing.

The principal audience for this document is intended to be people with an understanding of the basic concepts of computer science, including high-level language programming, operating systems, and computer architecture.

We would like to thank Dr. Russ Miller, Assistant Professor in the Department of Computer Science at the State University of New York at Buffalo, for several fruitful discussions and for reviewing this document.

CHAPTER 1

TERMINOLOGY

Introduction

This chapter serves to familiarize the reader with basic parallel processing terminology and concepts. Since the field of parallel processing is such a new field, terminology is often inconsistent; a variety of definitions can be found in the literature for any given parallel processing concept. Therefore, we recommend all readers, both knowledgeable and new to the field of parallel processing, to familiarize themselves with the particular set of definitions adopted throughout this document.

1-1 Supercomputers

Supercomputers are defined as the most powerful general-purpose scientific computer systems available at a given time. Notice that according to this definition, supercomputers have always, and will always exist. [Lincoln 1982]

[Kozdrowicki Theis 1984] states that a supercomputer is generally characterized by three main features: (1) high computational speed, (2) large main memory, and (3) fast and large secondary memory.

[Kosinski 1987] states that to qualify as a *minisupercomputer*, at the time of the writing of this document, a system must meet the following three criteria: (1) it must perform at least some scientific and engineering applications using 64-bit floating point arithmetic at a peak speed of not

less than one-tenth that of a low-end supercomputer, (2) it must be capable of running an entire compiled program, and (3) the typical price should range between $200,000 and $1 million, with the maximum price for a fully configured system at about $2 million. Common minisupercomputer features include large memories and some combination of vector or parallel support (to be defined).

[Hwang 1987] classifies supercomputers as either full-scale supercomputers, near supercomputers or minisupercomputers according to performance and cost. Figure 1-1 illustrates the performance and cost ranges of these three classes of commercial supercomputers in 1987.

	Class	Peak Speed	Cost	Examples
SUPERCOMPUTERS	full-scale supercomputers	200 MFLOPS to 2.4 GFLOPS	$2M to $25M	Cray 2, Cray X-MP, NEC SX, FACOM VP, ETA-10, IBM GF11
	near supercomputers	50 MFLOPS to 500 MFLOPS	$1M to $4M	Loral MPP, CDC Cyberplus, BBN Butterfly, Connection Machine
	minisupercomputers	10 MFLOPS to 100 MFLOPS	$100K to $1.5M	Alliant FX/8, CMU Warp, Convex C-1, SCS-40, ELXSI 6400, Intel iPSC, Encore Multimax
	superminicomputers (not considered a supercomputer)	Less than 0.5 MFLOPS	$20K to $400K	VAX/780, VAX 8600, IBM 4300, IBM 9370

Figure 1-1. Supercomputer classes and performance/cost ranges in 1987. The superminicomputer category is only given as a comparison. Super-minicomputers are **not** supercomputers.

1-2 Parallel Processing

[Quinn 1987] points out that while most high-performance modern computers exhibit a great deal of concurrency, it is not desirable to call every modern computer a parallel computer. The concurrency of many machines is totally invisible to the user. For this reason, we adopt the following definitions.

Parallel processing is a type of information processing that emphasizes the concurrent manipulation of data elements belonging to one or more processes solving a single problem. Pipelining and parallelism are two ways of achieving concurrency. Pipelining increases

concurrency by dividing a computation into a number of steps, while parallelism is the use of multiple resources to increase concurrency. A *parallel computer* is a computer designed for the purpose of parallel processing. [Quinn 1987]

1-3 Pipelining

A *pipelined computation* is a computation divided into a number of steps, called *stages*, which can operate autonomously, and the stages are then performed in overlapped mode. Each stage works at full speed on a particular part of the whole computation. The output of one stage is the input to the next stage. Assuming all stages work at the same speed, the work rate of the pipeline is equal to the sum of the work rates of the stages (once the pipe is full). Concurrency is increased as a result of dividing the computation into a number of steps. [Quinn 1987]

In *instruction pipelining*, the execution of each instruction is divided into a number of stages, such as instruction fetch, instruction decode, operand fetch, and instruction execute. By using pipelining, more than one instruction can be in some stage of execution at the same time. [Perrott Zarea-Aliabadi 1986]

Pipelining the data stream (*data pipelining*) is a natural evolution from the traditional serial model of computation. Instead of fetching scalars from memory and performing arithmetic on them, vectors are streamed from memory into the CPU, where pipelined arithmetic units manipulate them. [Quinn 1987]

Figure 1-2 illustrates the relationship between sequential and pipelined execution based on a four-stage process with each stage taking one unit of time.

1-4 Serialism versus Parallelism versus Pipelining

Suppose it takes 4 units of time for a lawn service to do one lawn and that there are four steps to the process - *rake, mow, edge, and sweep* - each requiring exactly one unit of time. A single person can do a lawn by spending one unit of time raking, one unit of time mowing, one unit of time edging and one unit of time sweeping. Therefore, a single person can do one lawn in 4 units of time, two lawns in 8 units of time, three lawns in 12 units of time, and so on, as shown in Figure

1-3a. Now suppose each of the four subtasks is assigned to a different person, Alex, Bob, Chris, and Denise. Alex rakes a lawn every time unit and then gives Bob the okay to start mowing. After Bob is through, he gives Chris the okay to start edging. After Chris is through, she gives Denise the okay to start sweeping. As each person completes their task on the current lawn, they move onto the next one. By pipelining the process, one lawn is completed in 4 units of time (initial time to fill the pipeline), two lawns are completed in 5 units of time, three lawns are completed in 6 units of time, and so on, as shown in Figure 1-3b. Now suppose the said lawn service is doing so well that they are able to hire three more lawn crews, making a total of four crews. Each crew does one complete lawn and then moves onto the next one. Clearly, four lawns are done in 4 units of time, eight lawns are done in 8 units of time, twelve lawns are done in 12 units of time, and so on, as shown in Figure 1-3c.

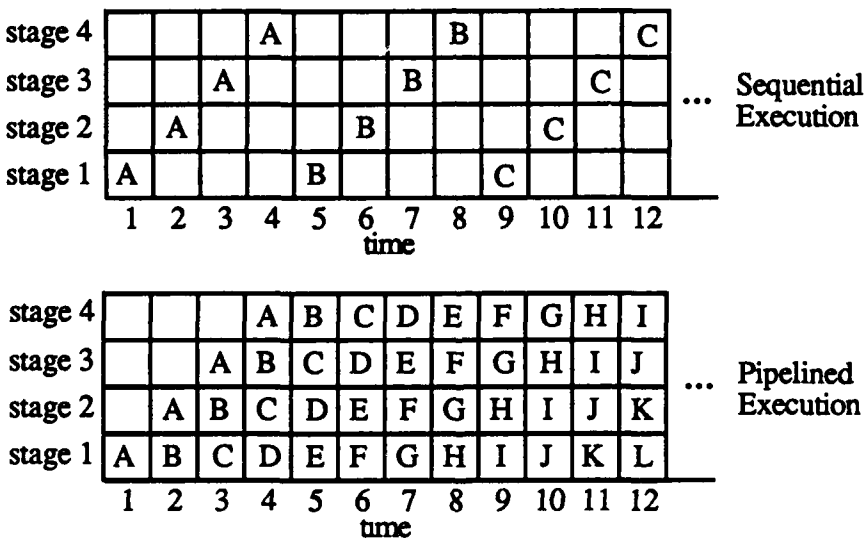


Figure 1-2. An illustration of the relationship between sequential and pipelined execution of a process that has four stages, with each stage of the process taking one unit of time.

1-5 Vector Computers

In contrast to the instruction set of a serial computer which allows the manipulation of only scalar operands, the instruction set of a *vector computer* contains operations on vectors as well as scalars. [Quinn 1987] The computational processes of a vector computer are pipelined.

In general, tasks are usually divided among a vector processor and a very high speed scalar

processor since it is unlikely that all of the code in a particular program will be vectorizable. The high speed scalar processor is used to avoid a system bottleneck or the degradation of vector performance. [Perrott Zarea-Aliabadi 1986]

Vector computers can be divided into two categories, *memory-to-memory* vector computers and *register-to-register* vector computers. In the memory-to-memory architecture, source operands and intermediate and final results are retrieved directly between the pipelines and the main memory. In the register-to-register architecture, operands and results are loaded from the main memory to a large number of vector or scalar registers before they can be used by the pipelines. [Tutorial 1984]

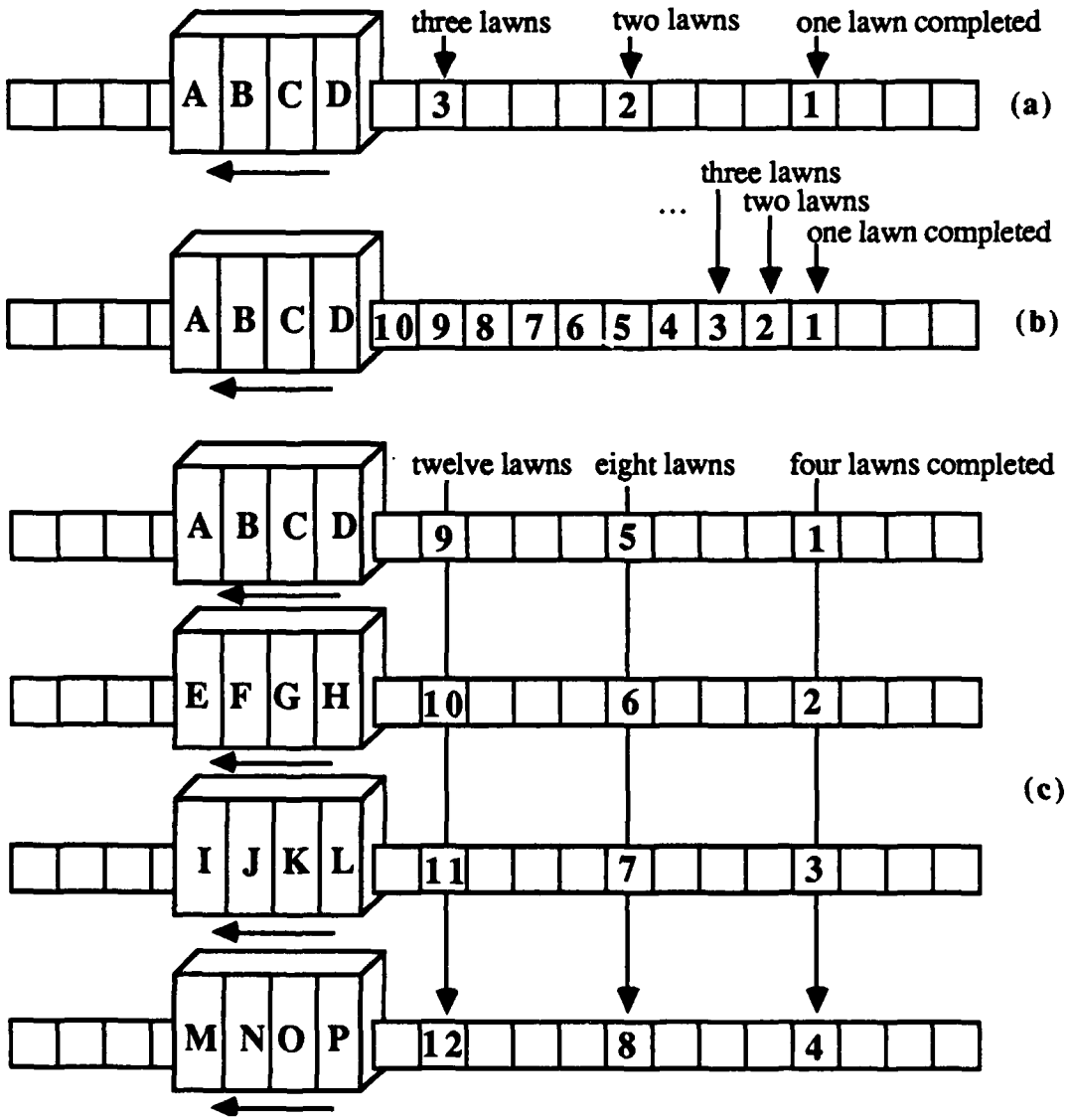


Figure 1-3. (a) Serialism vs. (b) Pipelining vs. (c) Parallelism. (Each square represents one time unit.)

Vector Terminology and Concepts

The primary reference for this section is [Quinn 1987], except as noted.

A *vector* is an ordered *n*-tuple of elements. For example, a row or column of a two-dimensional array is a vector. Vectors are made up of scalar quantities such as integers, floating-point numbers, booleans, or characters. In general, vector instructions can be divided into three categories as shown in Figure 1-4.

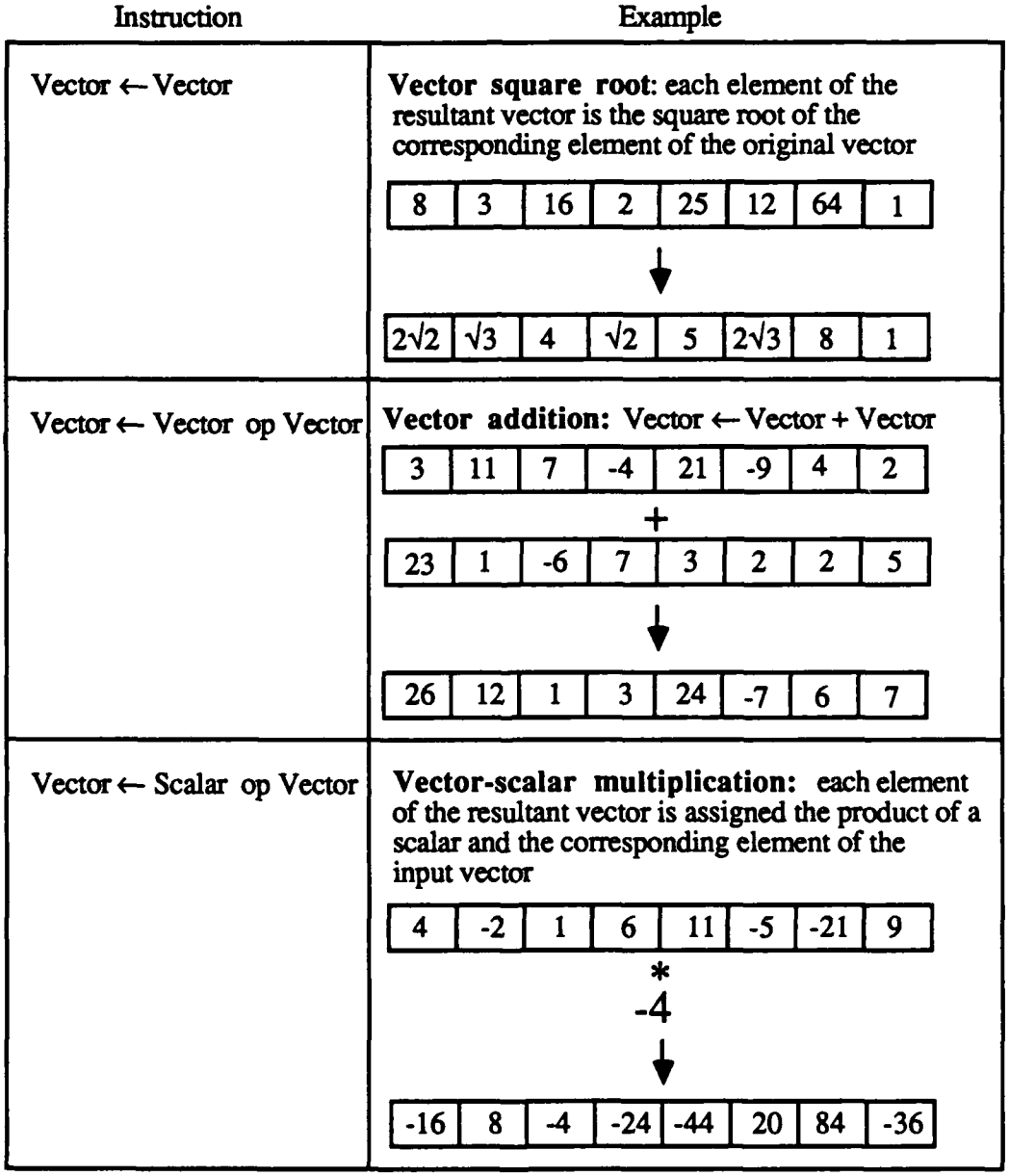


Figure 1-4. Three kinds of vector instructions. (Note that an instruction of the form Scalar ← Vector (e.g., vector summation) is not considered to be a vector instruction. This operation has a scalar result, and it cannot be pipelined.)

Masking vectors are boolean vectors used to enable or disable operations on particular vector elements. Figure 1-5 illustrates the use of a masking vector to compute the absolute value of a vector. When an operation is to be performed on a very small percentage of the elements of a vector, it is not a good idea to use a masking vector because it is wasteful to manipulate a long vector when so little work is done. The *compress* operation provides an alternative. A compress instruction loads a vector according to the values in a corresponding masking vector, at which point the compressed vector can be manipulated as an operand in vector instructions. The inverse of the compress operation is an *expand* operation. The expand operation is used to store a vector according to the values in a corresponding masking vector. Figure 1-6 illustrates both compress and expand operations. One final example of a vector operation is the merge operation, which merges two vectors according to the values in a masking vector. Figure 1-7 illustrates the merge operation.

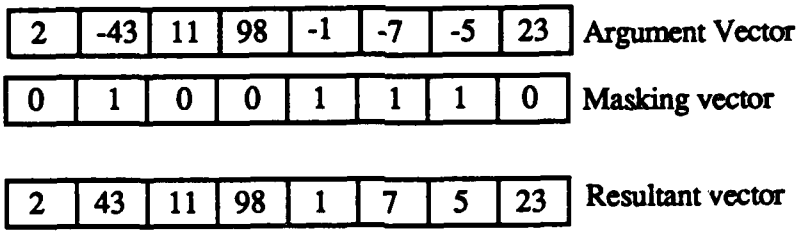


Figure 1-5. Use of a masking vector to compute the absolute value of a vector by negating selected elements.

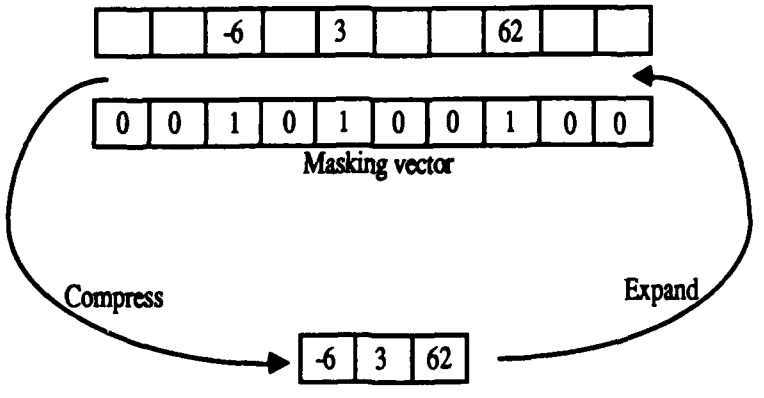


Figure 1-6. Illustration of compress and expand vector operations.

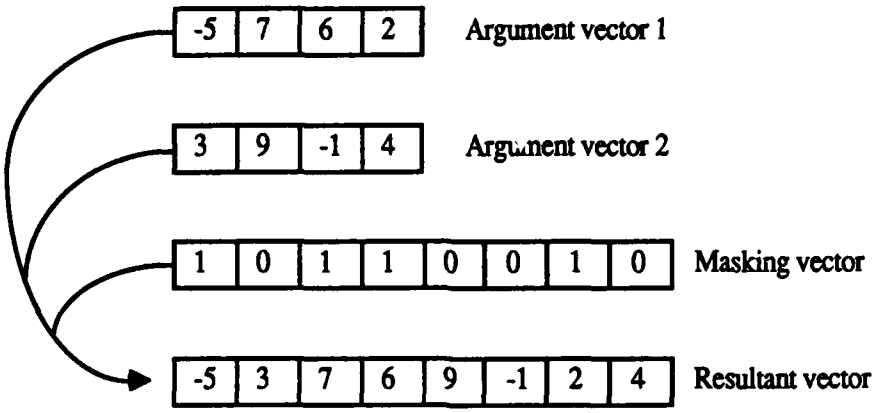


Figure 1-7. Illustration of a merge vector operation. A 1 in the masking vector means take the next argument from vector 1; a 0 means take the next argument from vector 2.

The compiler's role in analyzing user programs and producing object codes to execute on the vector hardware is called *vectorization*. The portion of the compiler which carries out the vectorization is called the *vectorizer*. The process of modifying and adapting an application program in order to reveal its vector content and to improve the performance is called *vector migration*. Vector migration assists the vectorizing compiler in exploiting the vector hardware for that program. [Hwang 1987]

The basic unit of vectorization is the do loop. *Strip mining* is the process of dividing a do loop into a number of smaller loops to be executed sequentially. This process is performed in register-to-register pipelined vector computers because of the limitations of vector registers. For example, suppose a program written for the Cray-1 (a register-to-register architecture) contains a single Fortran do loop that performs vector addition on two 100 element vectors. The vector registers on the Cray-1 can hold at most 64 elements; so the do loop must be performed in two steps: (1) load the first 64 elements from each vector, perform the 64 additions, and store the first 64 results; and (2) load the final 36 elements from each vector, perform the final 36 additions, and store the final 36 results. A series of vector operations can be *chained* together, allowing a second vector operation to begin as soon as results begin streaming out of the first functional unit's pipeline.

1-6 Shared Memory versus Distributed Memory

A *shared memory machine* has a single global memory accessible to all processors. A key feature of current shared memory systems is that the access time to a piece of data is independent of the processor making the request. In *distributed memory machines*, each processor has its own local memory. The only way for an application to share data among the processors is for the programmer to explicitly code commands to move data from one processor to another. The time it takes for a processor to access data is dependent on its distance from the processor that currently has the data in its local memory. [Karp 1987]

1-7 Flynn's Taxonomy

Michael Flynn [Flynn 1966] classifies computer architectures according to the concepts of instruction stream and data stream. An *instruction stream* is a sequence of instructions performed by a computer. A *data stream* is a sequence of data used to execute an instruction stream. An architecture is categorized by the multiplicity of hardware used to manipulate instruction and data streams. Given the possible multiplicity of instruction and data streams, there are four possible classes of computers: (1) Single-Instruction stream, Single-Data stream (SISD), (2) Single-Instruction stream, Multiple-Data stream (SIMD), (3) Multiple-Instruction stream, Single-Data stream (MISD), and (4) Multiple-Instruction stream, Multiple-Data stream (MIMD). [Quinn 1987] These four classes of computers are described below.

Single-Instruction stream, Single-Data stream (SISD) Computers

In *single-instruction stream, single-data stream* computers, the execution of instructions may be pipelined, but only one instruction is decoded per unit time. SISD computers may have multiple functional units, but they are under the direction of a *single* control unit. Most serial computers fall under this category. [Quinn 1987]

Single-Instruction stream, Multiple-Data stream (SIMD) Computers

A *single-instruction stream, multiple-data stream* machine typically consists of n processing

elements (PEs), a control unit, and an interconnection network. The control unit stores the program and broadcasts instructions to all PEs simultaneously. Enabled PEs execute the same instruction at the same time, but on the contents of their own local memory. PEs may be enabled or disabled at any time during the execution of the program through the use of a *mask*. [Miller Stout 1987]

SIMD machines are generally designed to exploit the fine-grained parallelism of tasks such as those involving matrix operations and digitized pictures, where the same operation is performed on many different matrix or image elements. The word size that each PE in an SIMD machine operates on varies from system to system. For example, the Illiac IV uses 64-bit words and the Massively Parallel Processor (MPP) uses 1-bit words. As an example of the possible size of SIMD machines, the Connection Machine-2 has 65,536 simple processing elements. [Siegel 1984] SIMD models may differ from one another in that PEs may communicate with each other through shared memory or through some interconnection network. The shared-memory based SIMD model, shown in Figure 1-8, consists of a control unit, a global random access memory, and n processors. [Quinn 1987]

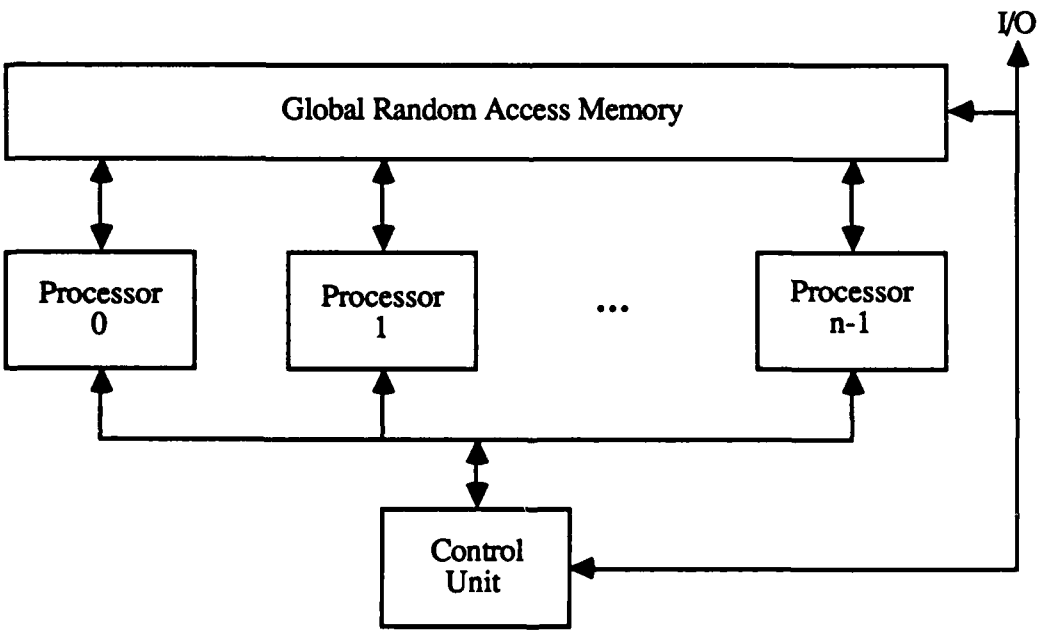


Figure 1-8. Shared-memory SIMD model diagram.

The majority of parallel SIMD algorithms being developed assume a shared global memory that allows the *n* PEs to access simultaneously any *n* locations in the entire memory space in constant time. This SIMD shared-memory model is the optimal parallel computer model. However, it is not feasible to build such machines for large numbers of processors, so no actual SIMD machines have been built based on this model. It is more realistic to assume that each PE has its own private memory and that PEs can pass data via an interconnection network, as shown in Figure 1-9. [Quinn 1987] (Interconnection networks are discussed in detail in Chapter 2.)

Note that it is still extremely important to develop algorithms for the optimal SIMD shared-memory model in order to find out what the absolute lower bounds are for solving problems. Once these optimal times are known, researchers can try to get as close as possible to these times on feasible architectural models.

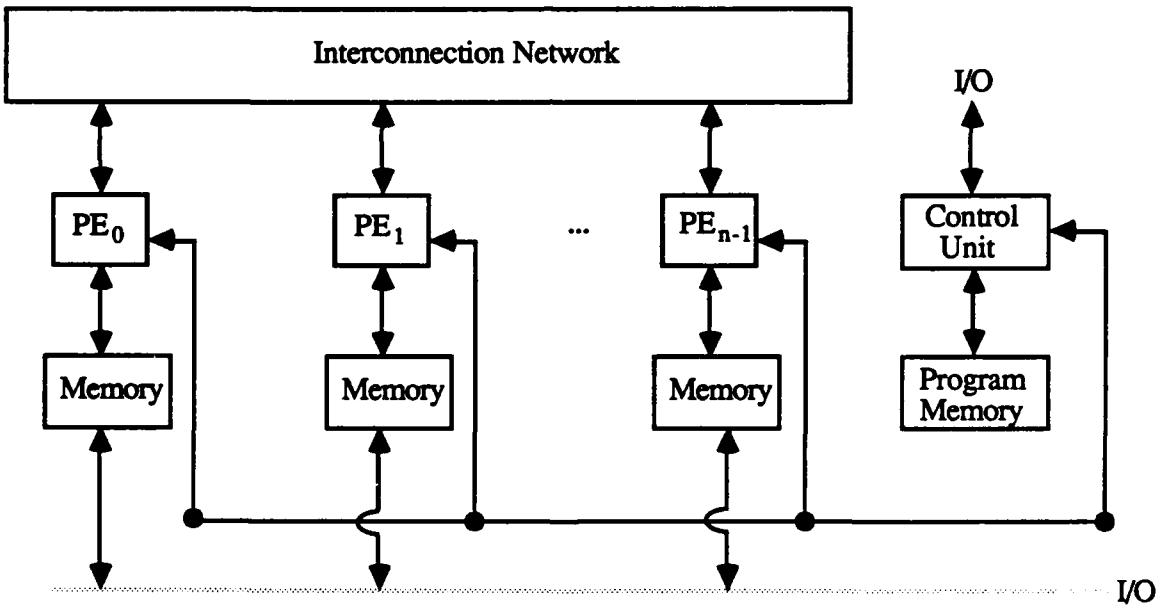


Figure 1-9. Interconnection network-based SIMD model diagram.

To understand better how SIMD machines operate, consider the following simple task (taken from [Quinn 1987]). Assume that each of *A*, *B*, and *Sum* is a one-dimensional array (vector) of *N* elements and that the task to be performed is the element-wise addition of *A* and *B*, storing the result in *Sum*. In a uniprocessor (serial) system, this task can be expressed as:

FOR I ← 0 TO N - 1 DO

$Sum[i] \leftarrow A[i] + B[i]$

This computation will take N steps on a serial machine because the body of the loop executes N times.

Now assume A, B, and *Sum* are stored in an SIMD machine with N PEs, such that $A[i]$, $B[i]$, and $Sum[i]$ are all stored in the local memory of PE i, $0 \leq i < N$. To perform an element-wise addition of the vectors A and B and store the result in *Sum*, all PEs would (simultaneously) execute

$Sum \leftarrow A + B$

with PE i doing the addition of $A(i)$ and $B(i)$, and storing the result in $Sum(i)$, for all i, $0 \leq i < N$ Since there are N PEs and N elements in each of the vectors A, B, and *Sum*, each PE does only one addition. The SIMD machine does in one step a task requiring N steps on a serial processor.

Consider a variation on this example (taken from [Siegel 1984]), which calls for use of the interconnection network. Assume the N-step serial task is:

FOR $i \leftarrow 1$ TO N - 1 DO
    $Sum(i) \leftarrow A(i) + B(i - 1)$
$Sum(0) \leftarrow A(0)$

Given the same data allocation as in the previous example, an SIMD machine performs this task in three different stages:

(1) The value of $B(i - 1)$ is moved through the interconnection network from PE_{i-1} to PE_i, $1 \leq i < N$. Most SIMD interconnection networks allow all of these PEs to do this in one parallel data transfer.

(2) In PE_i, add $A(i)$ to $B(i - 1)$ and store the result in $Sum(i)$, $1 \leq i < N$ (processor 0 is inactive).

(3) In PE_0, store $A(0)$ in $Sum(0)$ (all other PEs are inactive).

Multiple-Instruction stream, Single-Data stream (MISD) Computers

Multiple-instruction stream, single-data stream machines employ two or more processors that perform separate instructions on the same data. This approach is thought to be impractical and current literature states that no existing computers fall under this category.

Multiple-Instruction stream, Multiple-Data stream (MIMD) Computers

The following information is based on [Siegel 1984].

A *multiple-instruction stream, multiple-data stream* computer typically consists of *n* processing elements (PEs), *n* memory modules, and an interconnection network. Each of the *n* PEs stores and executes its own program. (Therefore, there are multiple instruction streams as opposed to the SIMD architecture that consists of only a single instruction stream.) Each PE fetches its own data on which it operate. (Thus, there are multiple data streams, as in the SIMD system.) The interconnection network provides communications among the processors and memory modules. While in an SIMD system all active PEs use the interconnection network at the same time (i.e., synchronously), in an MIMD system, because each PE is executing its own program, inputs to the network arrive independently (i.e., asynchronously).

MIMD machines can be organized as distributed memory machines or shared memory machines, illustrated in Figures 1-10 and 1-11, respectively. When using the shared-memory configuration, a local memory or cache can be associated with each processor.

Whereas all active PEs operate in a synchronous lockstep way in SIMD machines, PEs in an MIMD machine operate asynchronously with respect to each other. With this increased flexibility comes an increase in overhead costs to perform process synchronization and design programs for each of the *n* PEs (there may not be a single program, as in SIMD operation). Nevertheless, certain problems are not appropriate to the single instruction stream limitations of SIMD machines, so MIMD costs are justified.

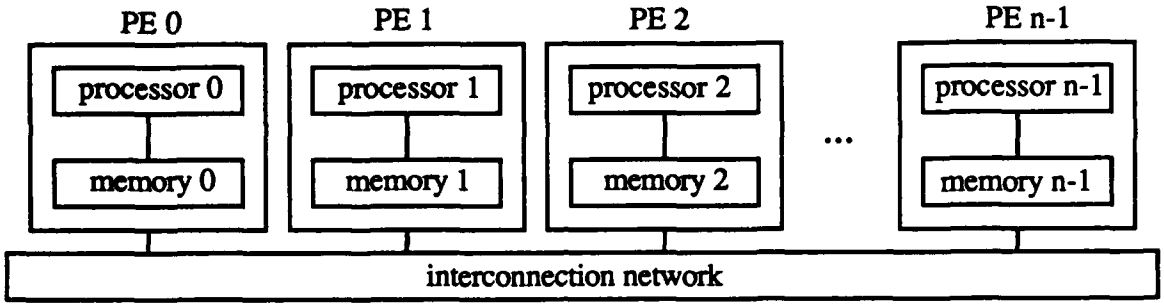


Figure 1-10. Distributed Memory MIMD machine configuration.

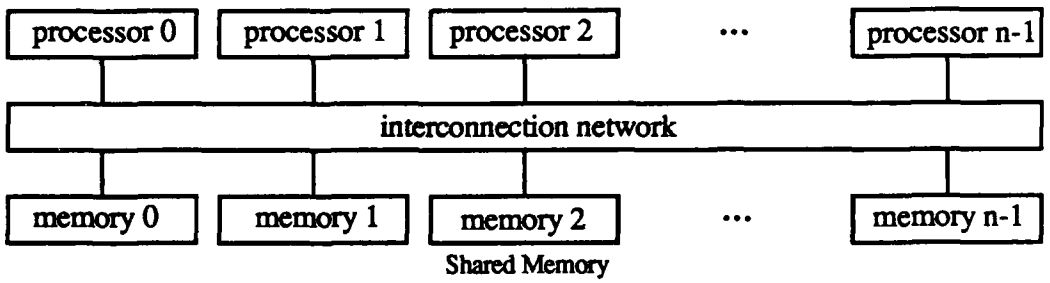


Figure 1-11. Shared memory MIMD machine configuration.

Multiple-SIMD Machines and Partitionable-SIMD/MIMD Machines

[Siegel 1984] discusses two variations on SIMD and MIMD machines, namely, multiple-SIMD machines and partionable-SIMD/MIMD machine. A *multiple-SIMD machine* is a parallel processing system that can be dynamically reconfigured to operate as one or more independent virtual SIMD machines of various sizes. A multiple-SIMD system consists of n processors, n memory modules, an interconnection network, and q control units, where $q < n$.

There are several advantages of multiple-SIMD systems over SIMD systems (with a similar number of processors). First, when high reliability is needed, numerous partitions can run the same program on the same data and compare results. Second, if a single processing element fails, only those partitions that include the failed processing element are affected. The rest of the system can continue to function. Third, multiple users can each be simultaneously executing a different SIMD program. Fourth, debugging is easier because the programmer can execute the program on, say, 32 processing elements, instead of 1024. Fifth, multiple-SIMD machines are more efficient because if a task requires only half of the processors, the other half can be used for another task. Finally,

two or more independent SIMD subtasks that are part of the same job can be executed in parallel, sharing results if necessary.

A *partitionable-SIMD/MIMD machine* is a parallel processing system that can be structured as one or more independent SIMD and/or MIMD machines of various sizes. A partitionable-SIMD/MIMD system consists of n processors, n memory modules, an interconnection network, and q control units, where $q < n$. Each processor can follow its own instructions (MIMD operation) in addition to being capable of accepting an instruction stream from a control unit (SIMD operation). The advantages listed for multiple-SIMD machines also apply here, where each partition can operate in either the SIMD or the MIMD mode of parallelism. In addition, in a partitionable-SIMD/MIMD machine the same set of processors can switch between the SIMD and MIMD modes of parallelism when performing a task.

1-8 Granularity

The *granularity* of a machine typically denotes the relative number and complexity of the processors. A *fine-grained* machine typically consists of a large number of small, simple (in terms of computational power and local memory) processors, while a *coarse-grained* machine typically consists of relatively few large, powerful processors. With respect to current technology (1987), fine-grained machines have on the order of 10,000 processors, while coarse-grained machines have on the order of 10 processors. *Medium-grained* machines represent a compromise in performance and size between that of fine-grained and coarse-grained machines, with on the order of 100 processors. In general, SIMD machines are thought of (and constructed) as fine-grained machines, where all processors operate in lockstep fashion on the contents of their own small local memory. MIMD machines are more often thought of as coarse-grained machines that either share a global memory or have the memory distributed among the processors. [Miller Stout 1987]

The term *granularity* is also used to express the ratio between computation and communication in a parallel program. Fine-grained parallel programs spend more time communicating than coarse-grained parallel programs. [Howe Moxon 1987]

1-9 Multiprocessors versus Multicomputers

Multiprocessors are MIMD machines that permit all processors to directly share main memory. In contrast, *multicomputers* are MIMD machines in which each processor has its own private memory and all communication and synchronization between processors is done through message passing. [Quinn 1987] A multiprocessor or multicomputer is further characterized by the topology of the interconnection network it uses. Multiprocessor organizations include the crossbar switch, the common bus to global memory, and the multistage network. Multicomputer architectures include topologies such as the hypercube, the mesh, the ring, and the tree. [Bhuyan 1987] Interconnection networks are discussed in detail in Chapter 2.

1-10 Tightly Coupled versus Loosely Coupled Machines

A computer is *tightly coupled* if the degree of interactions among the processors is high. Otherwise, a computer is considered *loosely coupled.* [Hwang Briggs 1984]

1-11 Processor Arrays and Associative Processors

A *processor array* consists of a single control unit and a set of identical synchronized processing elements (PEs) each of which has its own local memory. Instructions can be executed locally in the control unit or they can be broadcast to the PEs for execution in lockstep mode. No instruction can cause the PEs to perform dissimilar operations. The only possible variation in their operation is that they may be masked out under program control. [Perrott Zarea-Aliabadi 1986] A processor array can be classified as a special-purpose fine-grained SIMD machine.

An *associative processor* is a special kind of processor array. Whereas a processor array is built around a random access memory, an associative processor is built around an associative memory that allows the entire memory to be simultaneously searched for some specified contents. [Quinn 1987]

1-12 Attached Processors (array processors)

An *attached processor* (sometimes referred to as an *array processor*) is a special-purpose pipelined

processor which attaches to a general-purpose host computer and is designed to process large vectors or arrays. Attached processors are a low-cost alternative to pipelined vector computers. Similarities exist between attached processors and pipelined vector computers. Both contain multiple pipelined functional units and parallel data paths. Attached processors do not have vector instructions, but instead rely on carefully coded libraries of routines that use pipelining to achieve good performance on array and matrix manipulations. [Quinn 1987]

1-13 Dataflow Machines

The following information is based on [Barszoz Howard 1987], [Dennis 1984], [Ghosal Bhuyan 1987], and [Haynes Lau Siewiorek Mizell 1982].

Instruction execution in a *dataflow machine* is determined by the presence of data, not by a program counter. When all data for an instruction is present, it is executed without regard for its position in the program.

A dataflow program is represented by a *dataflow graph* showing data dependencies. A dataflow graph is composed of nodes and arcs. Nodes represent instructions to be executed. Arcs represent data dependencies between nodes. During execution, a node "fires," consuming input data and generating a result. *Tokens* carry copies of the result along the output arcs to dependent nodes. A node is enabled or ready to fire when there are tokens on all input arcs.

Consider the computation of the dot product of two vectors, $A = (a_1, a_2)$ and $B = (b_1, b_2)$, yielding $A \cdot B = a_1 b_1 + a_2 b_2$. The dataflow graph for this computation is given in Figure 1-12a. Each computation is encoded into an activity template as shown in Figure 1-12b. The dataflow program to be executed consists of activity templates, each with a unique address. When the required data is available for an activity template, the unique address of the template is enqueued, assigned a processor (as one becomes available), and executed. This generates one result packet for each destination field of the operation packet.

There are two basic implementations of dataflow: *static dataflow* and *dynamic dataflow*. Static dataflow allows at most one token on an arc at any given instant. A dynamic dataflow machine uses tagged tokens permitting more than one token to co-exist in any arc at any time. This tagging is

achieved by attaching a label with each token. The label essentially identifies the context in which the particular token was generated.

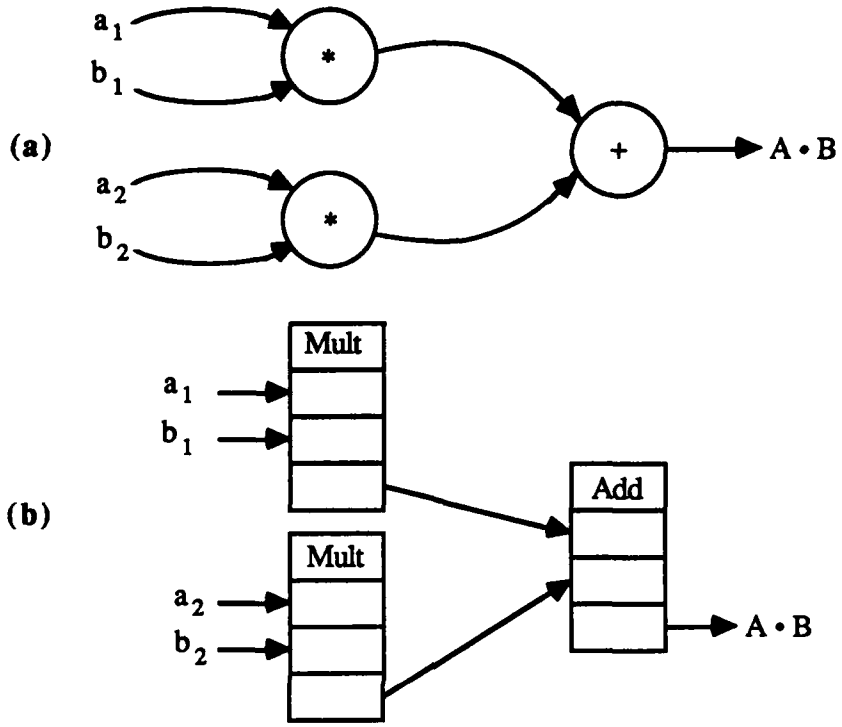


Figure 1-12. (a) The flow graph for the dot product of two vectors. (b) Corresponding activity templates.

1-14 Systolic Systems

The following information is based on [Haynes Lau Siewiorek Mizell 1982] and [Kung 1982].

A *systolic system* consists of a set of interconnected processing elements (PEs), each capable of performing some simple operation. Because simple and regular communication and control structures have substantial advantages over complicated ones in design and implementation, PEs in a systolic system are typically interconnected to form a systolic array or a systolic tree. Information in a systolic system flows between cells in a pipelined fashion, and communication with the outside world occurs only at the "boundary PEs." For example, in a systolic array, only those PEs on the array boundaries may be I/O ports for the system.

The basic principle of a systolic architecture is that by replacing a single PE with an array of PEs, a higher computation throughput can be achieved without increased memory bandwidth

(Figure 1-13). Data is "pulsed" through the array of PEs from the memory. Once a data item is brought out from memory it can be used effectively at each PE it passes while being "pumped" from PE to PE along the array.

Figure 1-15 illustrates multiplication of two band matrices, $C \leftarrow A \bullet B$, on a hexagonally connected systolic array. A matrix of bandwidth w may have ω diagonals that are not zeroes. Matrices A, B, and C are shown in Figure 1-14. Both matrix A and matrix B have bandwidth 4 along their principal diagonals. Thus, C has bandwidth 7 along its principal diagonal. A, B, and C are all of dimension $n \times n$. The entries outside the diagonal band are all zeroes. The first three iterations of the multiplication algorithm are given in Figures 1-16a, b, and c.

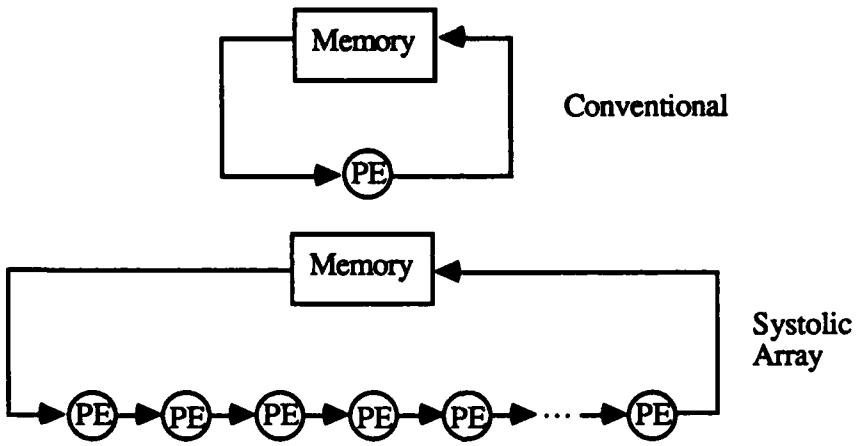


Figure 1-13. Illustration of the basic principle of a systolic architecture.

1-15 Reconfigurable Architectures

The objective of reconfiguration is to achieve the best possible match between a problem and the hardware. Reconfigurability is a broad term that is generally applied to features of a system that make it more flexible. Reconfiguration can take place in three places:

(1) *System level (includes the entire facility).* Reconfiguration is performed to keep the system running even in the presence of a failed subsystem or to "tune" the performance of a mixture of jobs in a multiprogrammed environment.

(2) *Within the machine itself.* Reconfigurability is important for achieving performance improvements and to a lesser degree, for fault tolerance. The areas most suitable for investigation

are reconfiguration of the processors and memory to achieve a better match to the program to be executed.

(3) *Hardware functional unit level.* Reconfiguration offers the potential to improve performance many times. The goal of this research is to develop design methods for computational units that have the performance of special purpose hardware but are not "hardwired". This involves the reconfiguration of data paths and control structures. [SRC 1986b]

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 & \cdots \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 & \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 & \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & & \\ 0 & 0 & a_{53} & a_{54} & \ddots & \ddots & \\ \vdots & & & & & & \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & b_{13} & 0 & 0 & 0 & \cdots \\ b_{21} & b_{22} & b_{23} & b_{24} & 0 & 0 & \\ 0 & b_{32} & b_{33} & b_{34} & b_{35} & 0 & \\ 0 & 0 & b_{43} & b_{44} & \ddots & \ddots & \\ 0 & 0 & 0 & b_{54} & & \ddots & \\ \vdots & & & & & & \end{bmatrix} =$$

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & 0 & 0 & \cdots \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & 0 & \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} & \\ c_{41} & c_{42} & c_{43} & c_{44} & \ddots & \ddots & \\ 0 & c_{52} & c_{53} & c_{54} & & \ddots & \\ \vdots & & & & & & \end{bmatrix}$$

Figure 1-14. Band matrix multiplication.

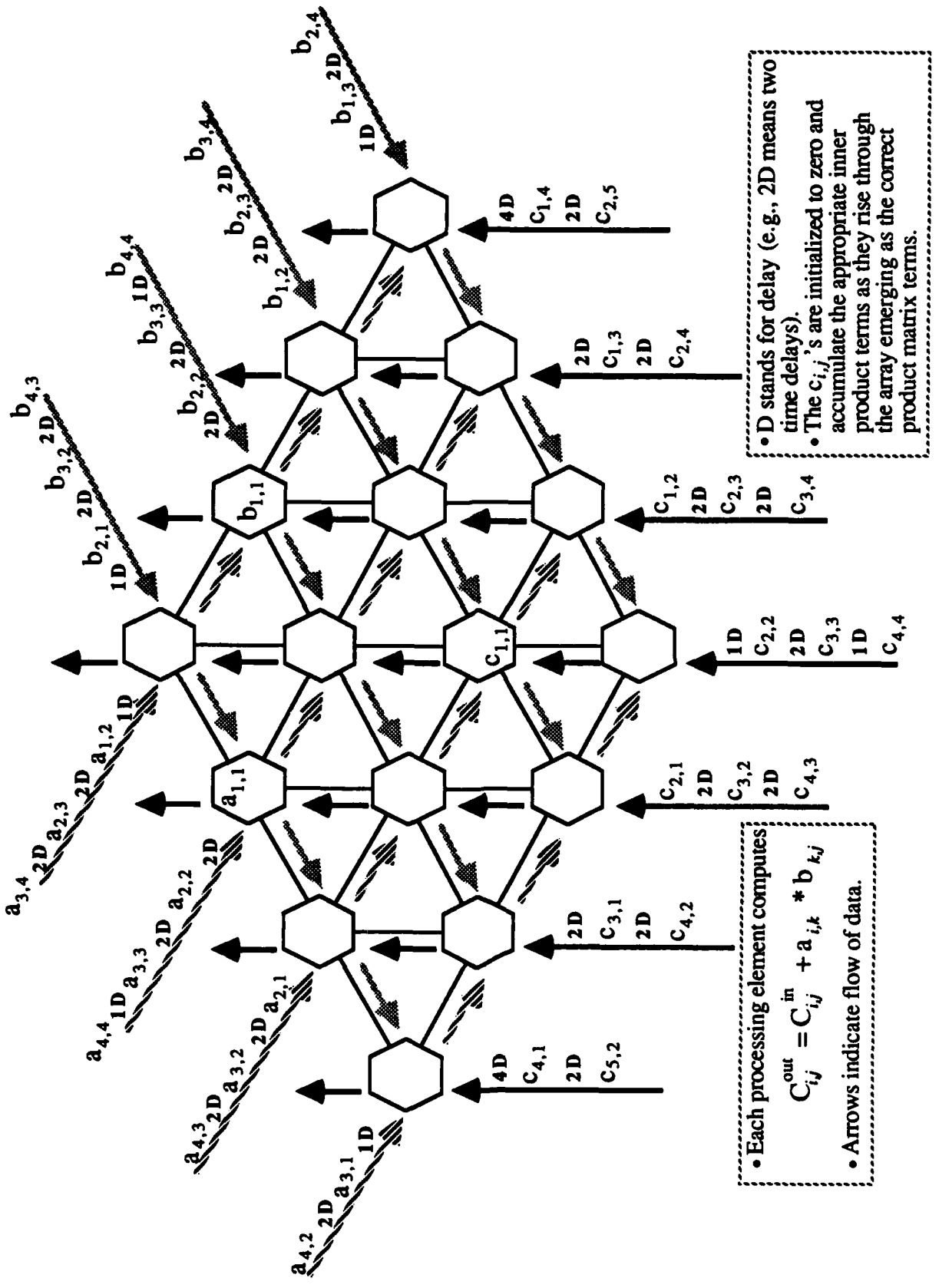


Figure 1-15. Illustration of band matrix multiplication on a hexagonally connected systolic array.

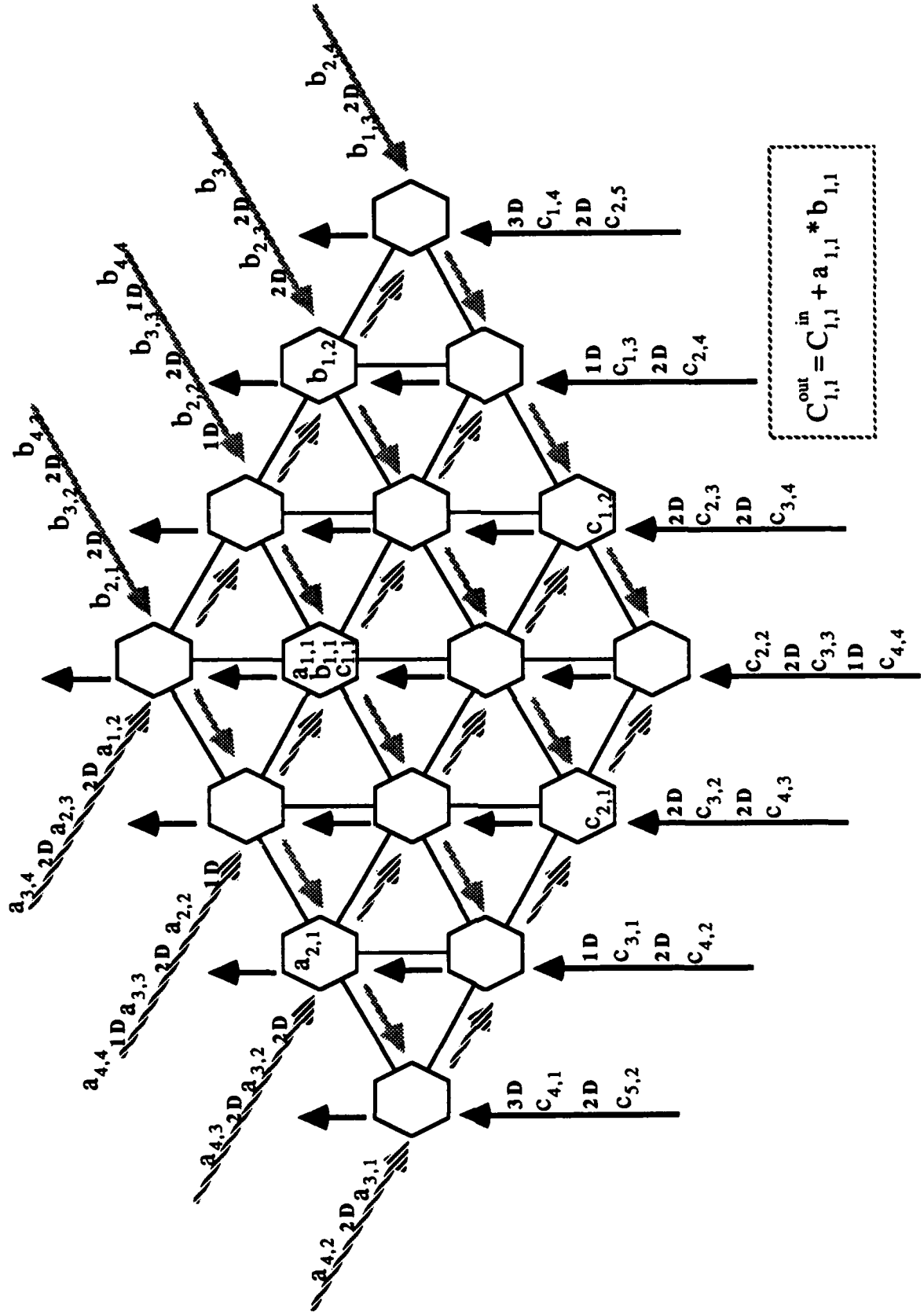


Figure 1-16a. The first iteration of the systolic array algorithm to multiply two band matrices.

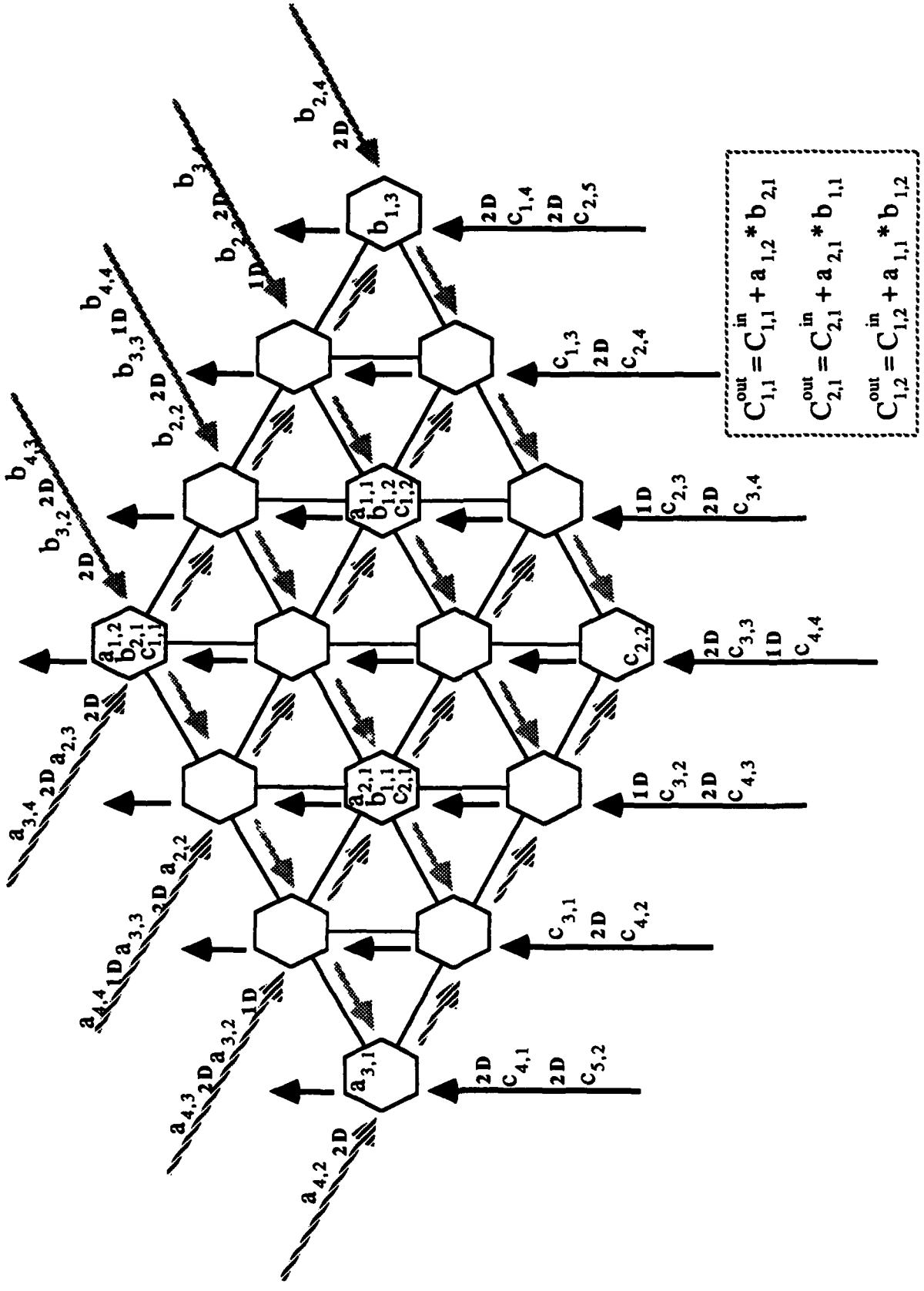


Figure 1-16b. The second iteration of the systolic array algorithm to multiply two band matrices.

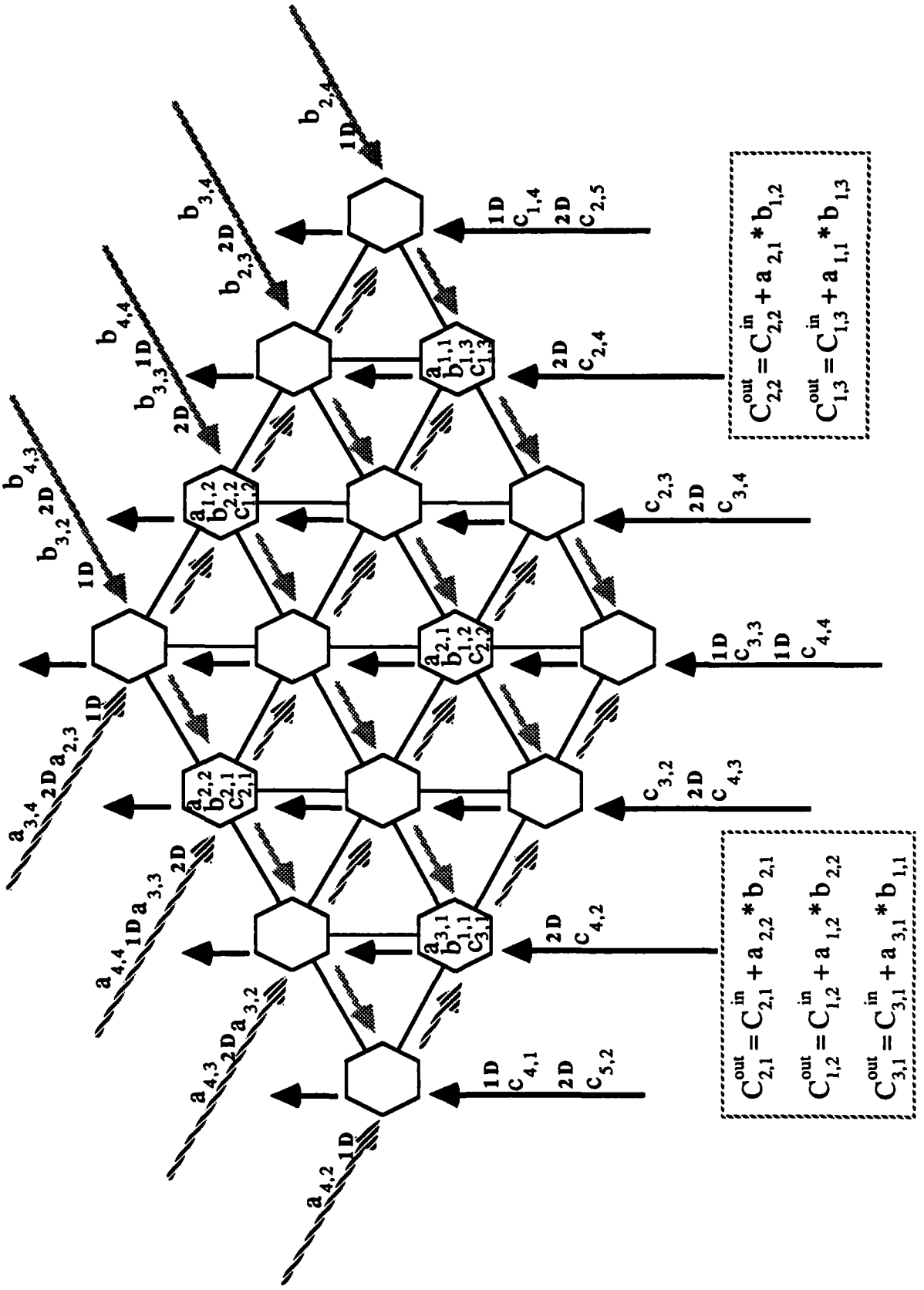


Figure 1-16c. The third iteration of the systolic array algorithm to multiply two band matrices.

The following information is based on [Quinn 1987].

A set of concurrent processes is said to be in *deadlock* if each holds nonpreemptible resources that must be aquired by some other process in order for it to proceed. Deadlock can occur in both multiprocessors and multicomputers. Figure 1-17 illustrates the idea of deadlock. Process A and Process B try to lock the same two resources. (The lock mechanism is discussed in Section 5-2.) Process A locks *resource1* while process B locks *resource2*. Process A is suspended when it trys to lock *resource2*. Likewise, process B is suspended when it trys to lock *resource1*. Neither process can proceed. They are in deadlock.

There are four conditions necessary for a deadlock to exist: (1) *mutual exclusion* - each process has exclusive use of its resource, (2) *nonpreemption* - a process never releases resources it holds until it is through using them, (3) *resource waiting* - each process holds resources while waiting for other processes to release theirs, and (4) *a cycle of waiting processes* - each process in the cycle waits for resources that the next process owns and will not relinquish.

There are at least three ways to address the problem of deadlock: (1) one can try to detect deadlocks when they occur and try to recover from them, (2) one can avoid deadlock by using beforehand information about requests for resources to control allocation so that the next allocation of a resource will not cause processes to enter a situation in which deadlock may occur, or (3) one can prevent deadlock by forbidding one of the first three conditions listed in the previous paragraph.

Process A	Process B
...	...
lock(*resource1*);	lock(*resource2*);
...	...
lock(*resource2*);	lock(*resource1*);
...	...

Figure 1-17. An illustration of a deadlock situation.

1-17 Speedup

In a parallel system consisting of a set of identical processing elements (PEs), the *speedup* of a program can be obtained by taking the ratio of the execution time on one PE to that of a number of PEs. If the number of PEs in the parallel system is n, then from the computational point of view the maximum speedup that the program can achieve is n. This is called *linear speedup*. However, there are many factors which can substantially reduce this ratio, including the amount of serialization in the program, overheads of communication/synchronization mechanisms, and system contention resulting from concurrent accesses to physical resources. [Hwang 1987]

1-18 MFLOPS, GFLOPS, and MIPS

MFLOPS is the acronym for millions of floating point operations per second, *GFLOPS* is the acronym for billions of floating point operations per second, and *MIPS* is the acronym for millions of instructions per second. The theoretical maximum speed for a given computer is called *peak speed.*

CHAPTER 2

INTERCONNECTION NETWORKS

Introduction

This chapter formally defines various *interconnection networks*, which are used to connect processors to processors and processors to memory. Specifically, we define an interconnection network to be a connection of switches and links that permits data communication between processors and/or memories in a system consisting of multiple processors. Many factors are involved in the cost-effectiveness of a particular network design, including the computational tasks it will be used for, the desired speed of interprocessor data transfers, the actual hardware implementation of the network, the number of processors in the system, and any cost constraints on the construction. [Bhuyan 1987]

The concept of switching methodologies for communications between processing elements may arise later in this document, so we define it here. The two major switching methodologies are *circuit switching* and *packet switching*. In circuit switching, a physical path is actually established between a source and a destination. In packet switching, data is put in a packet and routed through the interconnection network without establishing a physical connection path. In general, circuit switching is much more suitable for bulk data transmission, and packet switching is more efficient for many short data messages. [Hwang 1987]

2.1 Processor to Processor Interconnection Networks

A few definitions are necessary before specific processor to processor interconnection networks are discussed. The *degree* of a node is defined as the number of communication links per node. For example, the degree of a node in the interconnection network of Figure 2-1 is 7 (in general, n - 1). The *communication diameter* of a machine is the maximum of the minimum distance (number of communication links) between any two processors in the interconnection network. [Miller Stout 1987] For example, the maximum number of links a message must travel between any source node and any destination node along the shortest path in the interconnection network of Figure 2-1 is 1.

The communication diameter of a network is generally higher in machines connecting nodes of low degree than in machines connecting nodes of high degree. However, it is usually much more expensive to design networks with nodes of high degree. Ideally, low diameter machines consisting of nodes with low degree are desired.

We now describe a variety of interconnection networks. The first two show the extreme situations of node degree versus communication diameter.

Completely Connected Network

Ideally, each processor in a parallel computer should be linked directly to every other processor so that the system is *completely connected*, as illustrated in Figure 2-1. This topology is highly impractical when n is large because n - 1 unidirectional lines are required per processor. [Siegel 1984]

The completely connected network is an example of an interconnection network with high degree per node, in fact, the highest possible without duplicate connections, and with the lowest communication diameter, in that each node can communicate with every other node directly.

Ring Network

In a *ring network*, n processors are connected on a circular bus and each processor can communicate directly with its two nearest neighbors, as illustrated in Figure 2-2. [Haynes Lau Siewiorek Mizell 1982] Therefore, the degree of each node is at a minimum for a connected system

with more than two nodes. The maximum number of links which must be traversed to reach a destination node, however, might be as high as $n/2$.

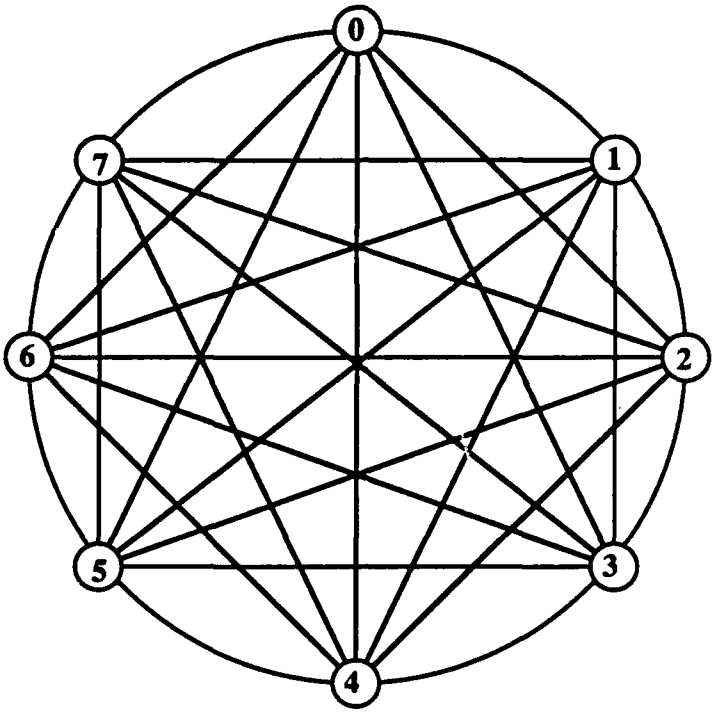


Figure 2-1. A completely connected network.

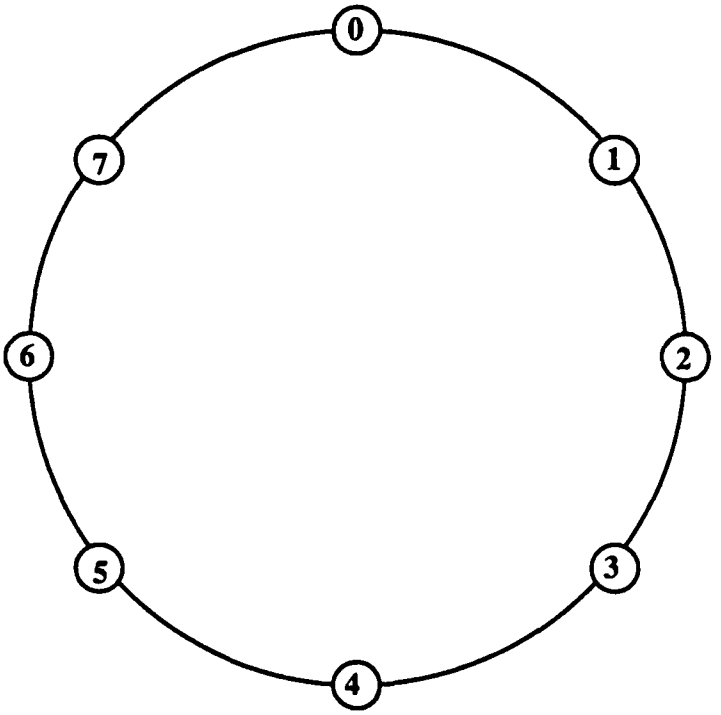


Figure 2-2. A ring network.

Mesh Network

The processors of a *mesh computer* of size n are configured as a rectangular lattice (Figure 2-3a). Meshes are frequently constructed as square lattices, where n is often an integral power of 4. Each processor in a mesh computer can communicate directly with its four nearest neighbors: north, south, east, and west. The communication diameter of a mesh with n processing elements is proportional to $n^{1/2}$, i.e., the edge length of the mesh.

Some variations on the mesh are derived from connecting the boundaries of the mesh to form a cylinder (north-south or east-west), torus (doughnut), or spiral. Figure 2-3b shows a variation allowing for wrap-around connections between processors on the edge of the mesh. Several interconnection networks have been designed which are augmentations to the mesh. These include the pyramid computer and the mesh-of-trees, both of which are discussed later in this Chapter.

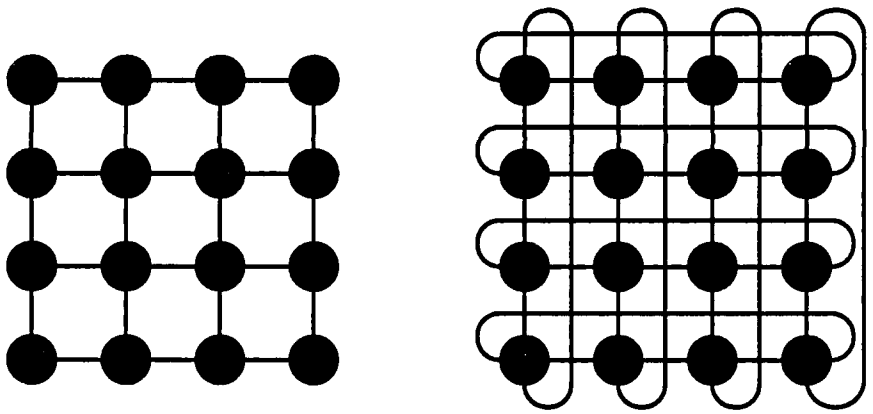


Figure 2-3. Two-dimensional mesh networks of size 16 (4×4): (a) No wrap-around. Interior nodes can communicate with 4 other nodes. (b) Wrap-around between nodes in the same row or column. All nodes can communicate with 4 other nodes.

Hypercube (Binary n-cube)

A *hypercube* of size n, where n is an integral power of 2, has n processing elements (PEs) indexed by the integers $\{0, \ldots, n - 1\}$. Viewing each integer in the index range as a $\log_2(n)$-bit string, two PEs are connected via a bidirectional communication link if and only if their indices differ by exactly one bit, as illustrated in Figure 2-4. The communication diameter of a hypercube of size n is

proportional to log(n), in that PE x can send a piece of data to PE y by correcting each of the differing bits in the source node's label to be the destination node's label, as illustrated in Figure 2-5. [Miller Miller 1987] [Hypercube 1986]

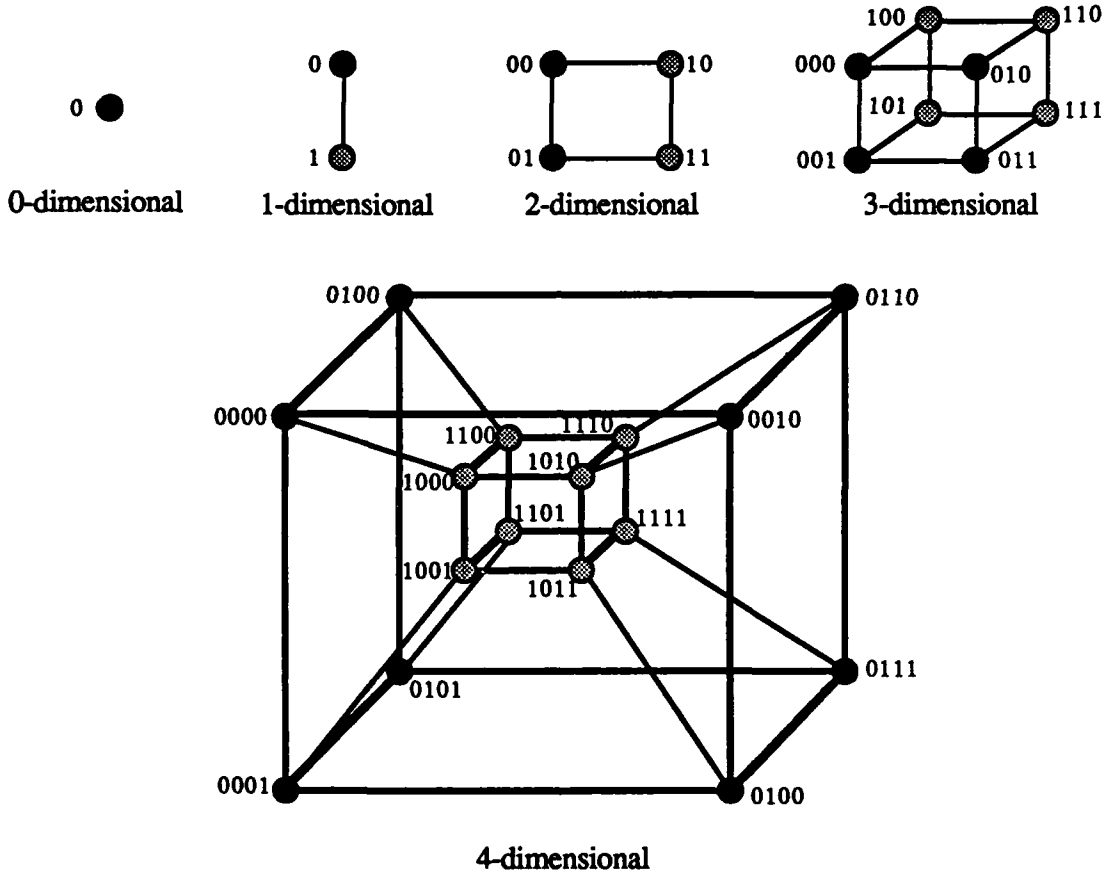
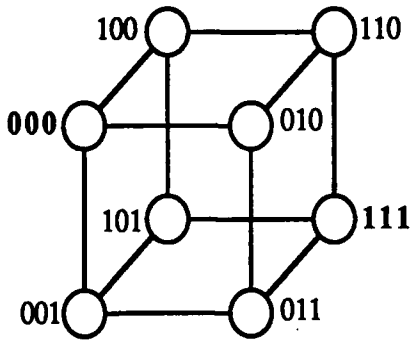


Figure 2-4. There are n = 2^d nodes in a d-dimensional hypercube. A hypercube of size n is created recursively from two hypercubes of size $n/2$ by labeling each hypercube of size $n/2$ identically and independently with the indices {0, ..., $n/2$ - 1}, and then appending a 1 in front of the bit-strings of one of the cubes and a 0 in front of the other, 'creating' a new link from each PE in one cube to the corresponding PE in the other cube.

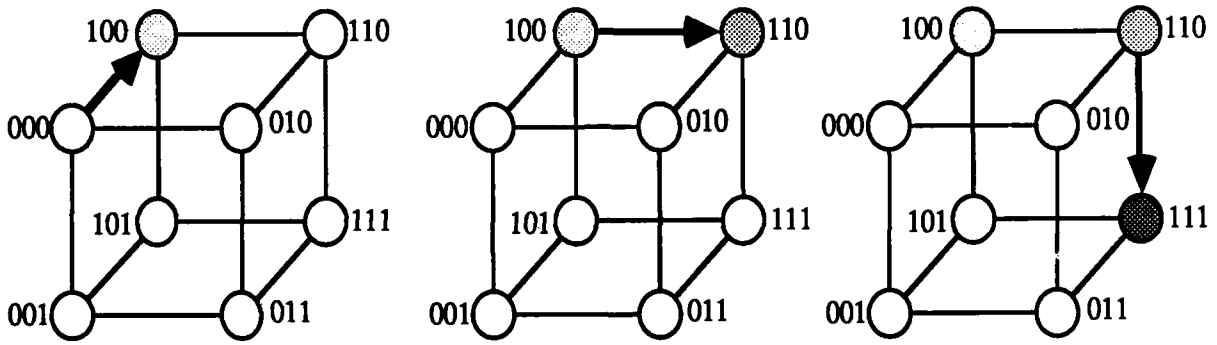
Cube-Connected Cycles Network

The *cube-connected cycles network* is obtained by taking a hypercube and replacing each of its n nodes with a ring of $d = \log(n)$ nodes, as illustrated in Figure 2-6. Each ring node connects to one of the d links incident on the vertex, fixing the degree of each node at three. The communication diameter of a cube-connected cycles network is the same as a hypercube network, but the cube-connected cycles network has an advantage over the hypercube network in that the degree of each node is fixed at three. Therefore, the cube-connected cycles network is expandable "forever."

The degree of a node in a hypercube network grows logarithmically with each new dimension, and therefore, is not expandable "forever." [Reed Grunwald 1987]



Send a message from node 000 to node 111.



Correct the high-order bit. Correct the middle bit. Correct the low-order bit.

Figure 2-5. Illustration of why the communication diameter of a hypercube is proportional to $\log(n)$. The nodes of a hypercube are labeled by the integers $\{0, \ldots, n\text{-}1\}$, each viewed as a $\log_2(n)$-bit string. To send a message from node x to node y, each differing bit in node x's label is corrected to form node y's label. For example, in this illustration, a message is sent from node 000 to node 111 via the path 000 → 100 → 110 → 111. Note that this is not the only possible path - five other paths exist.

Pyramid

A *pyramid* of size n is a machine that can be viewed as a complete 4-ary rooted tree of height $\log_4(n)$, with additional horizontal interprocessor links so that the processors in every tree level form a two-dimensional square mesh, as illustrated in Figure 2-7. There is a two-dimensional network of $n = k^2$ processing elements at the base of a pyramid of size n. The levels of the pyramid are numbered from 0 at the base of a pyramid to $\log_4(n)$ at the apex of the pyramid. A PE at level i is connected via bidirectional communication links to its nine neighbors (assuming that they exist):

a parent at level i - 1, four siblings at level i, and four children at level i + 1. The communication diameter of a pyramid computer of size n is proportional to $\log(n)$, in that any two PEs can communicate by sending information through the apex. [Miller Stout 1987]

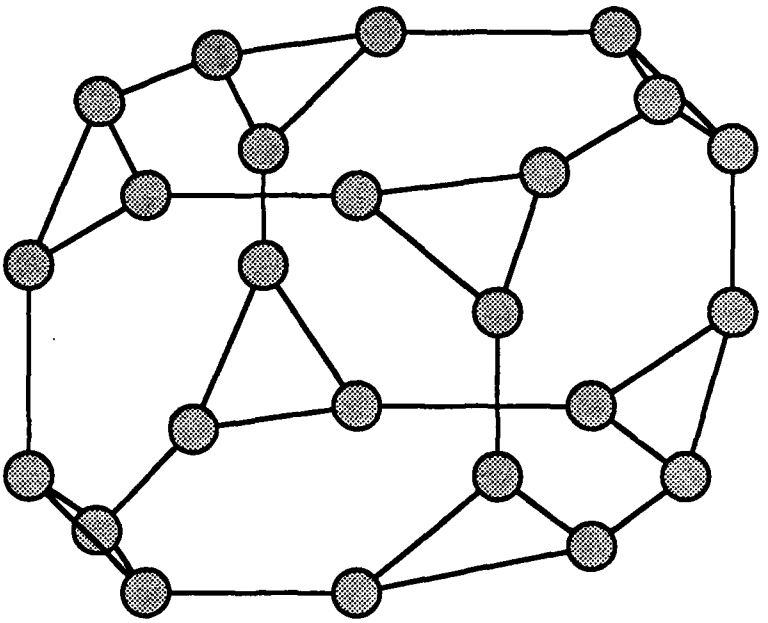


Figure 2-6. A cube-connected cycles network for $d = 3$.

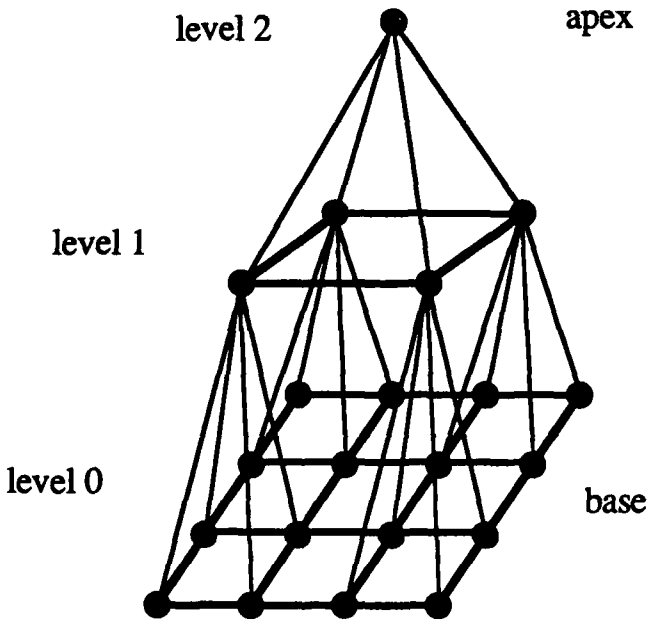


Figure 2-7. A pyramid of size $n = 16$.

Mesh-of-Trees

A *mesh-of-trees* of base size n, where n is an integral power of 4, has a total of $3n - 2n^{1/2}$ processing elements (PEs). n of these are base PEs arranged as a mesh of size n. Above each row and above each column of the mesh is a perfect binary tree of PEs. Each row (column) tree has as its leaves an entire row (column) of base PEs. All row trees are disjoint, as are all column trees. Every row has exactly one leaf PE in common with each column tree. The communication diameter of a mesh-of-trees of size n is proportional to $\log(n)$, in that two PEs can communicate by a row and column tree. A sample mesh-of-trees is given in Figure 2-8. [Miller Stout 1987]

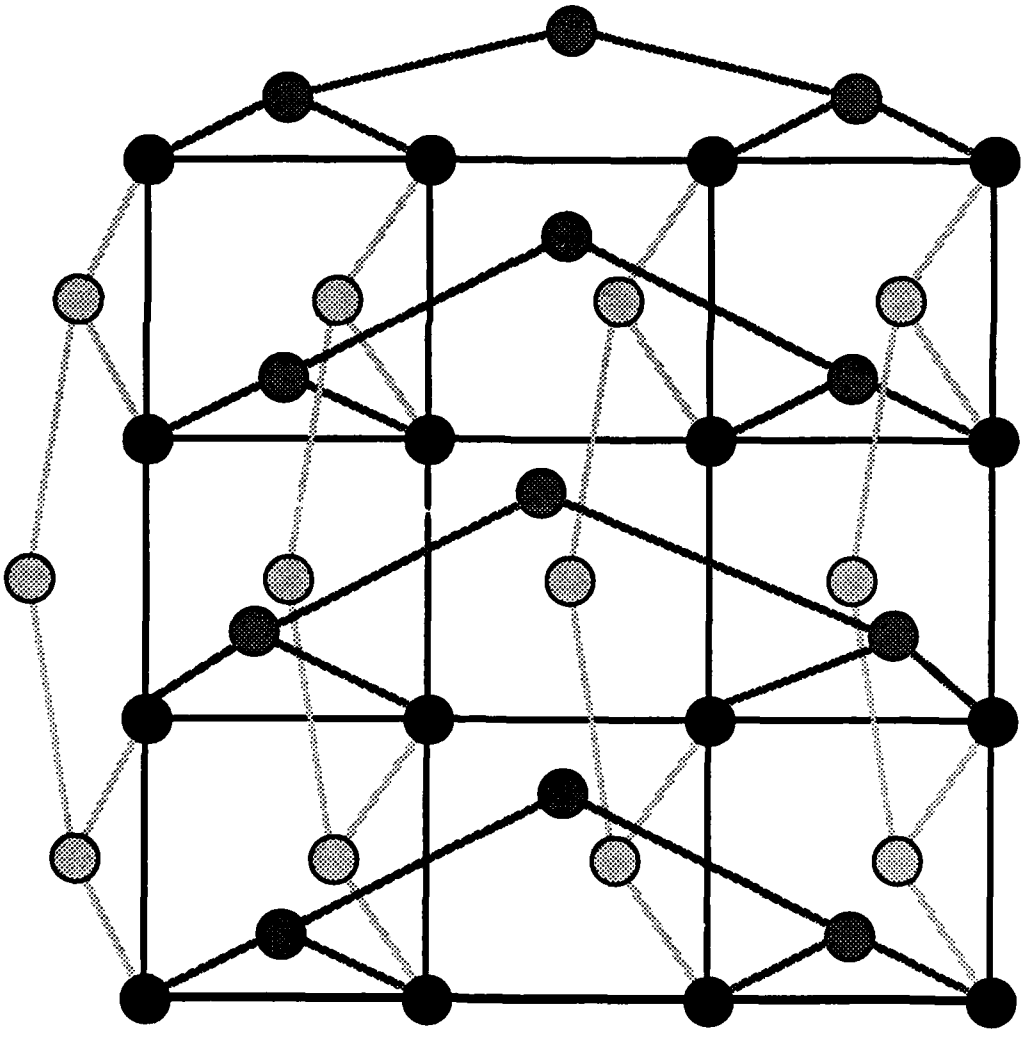


Figure 2-8. A mesh-of-trees of base size $n = 16$.

Shuffle-Exchange Network

A *shuffle-exchange network* consists of $n = 2^k$ nodes, labeled from 0 to n - 1 and two kinds of connections, shuffle and exchange, as illustrated in Figure 2-9. Exchange connections link pairs of nodes whose labels differ in their least significant bit. The perfect shuffle connection links node i with node $2i$ modulo n - 1, with the exception that node n - 1 is connected to itself. [Quinn 1987]

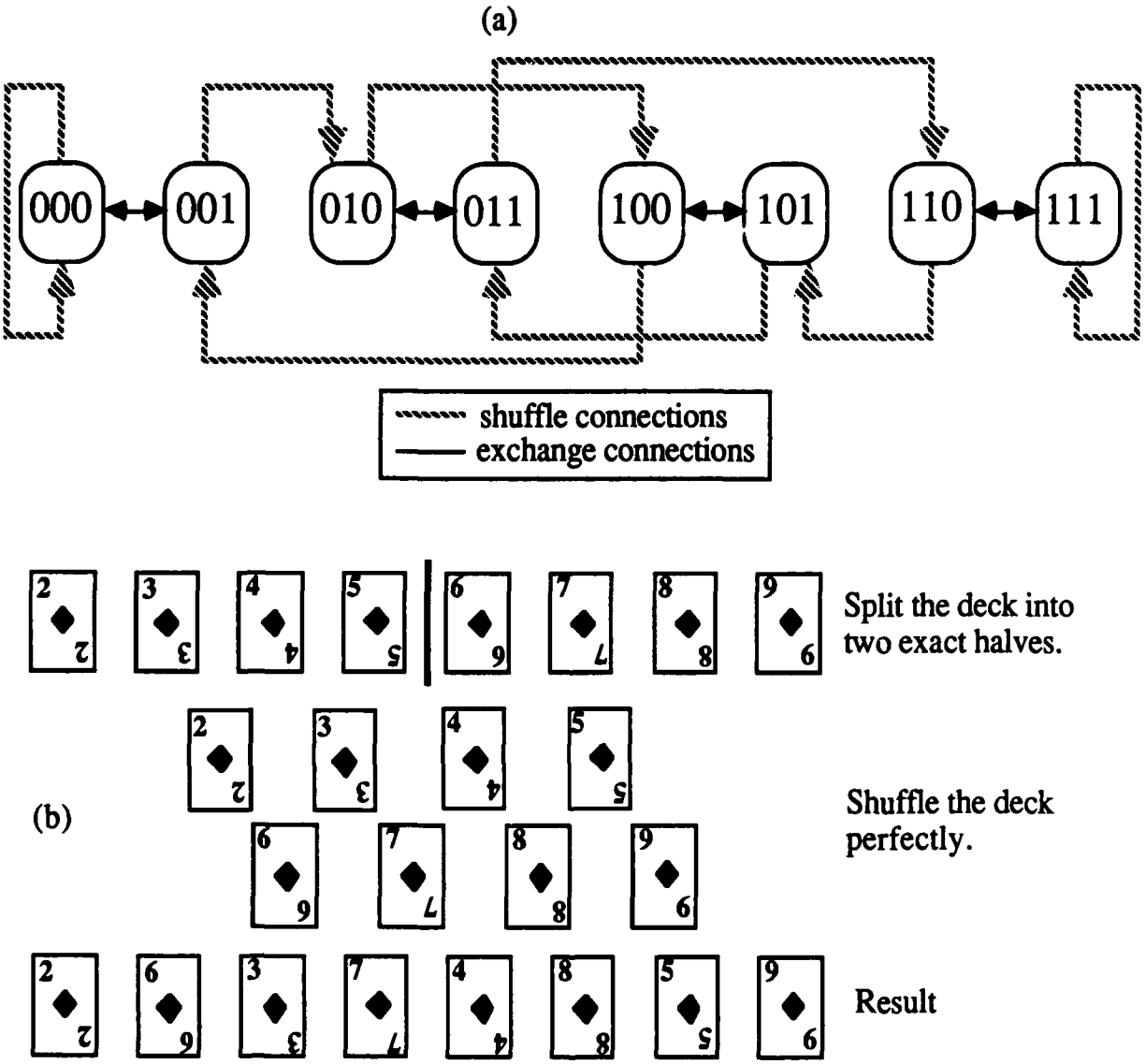


Figure 2-9. (a) A shuffle-exchange network with 8 nodes. (b) Illustration of where the name *perfect shuffle* comes from: notice that the final position of the card that began at index i can be determined by following the shuffle link from node i in the shuffle-exchange network of (a).

X-Tree

An *X-tree* is a simple binary tree with all nodes at each level connected in a ring (Figure 2-10), reducing the communication bottleneck near the root of the tree. [Reed Grunwald 1987]

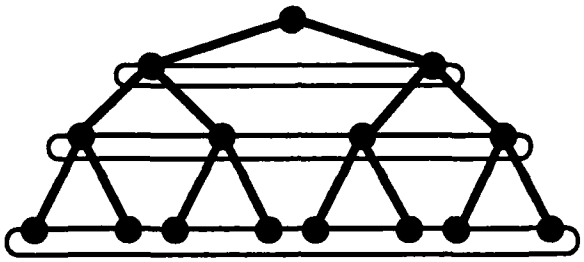


Figure 2-10. An X-tree.

Star Network

In a *star network*, a central processor connects to all other processors (Figure 2-11). The maximum distance from one processor to another is two. The center processor usually differs from the other processors because of the huge amount of traffic that it must handle. This traffic problem limits the number of processors feasible in such a system. [Karp 1987]

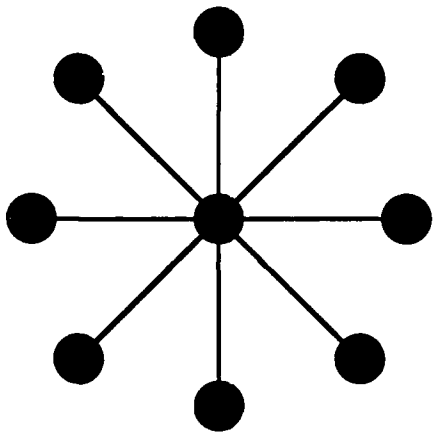


Figure 2-11. A star network.

2-2 Processor to Memory Interconnection Networks

We now describe a variety of interconnection networks which are used to connect processors to memories.

Single Shared Bus

Although the *single shared bus* (Figure 2-12) is the least complicated interconnection network, it has the disadvantage that only one processor can access the shared memory at a time. Large systems of this type are impractical to build because only so many processors can share a bus before the bus becomes saturated. [Siegel 1984]

The performance of a bus-based system can be improved by adding high-speed local memories (*caches*) to each processor. Caches can respond to most memory references, reducing contention for the common bus. However, cache consistency becomes a problem because each processor can modify its copy of data elements independently. Special hardware can be added to synchronize the contents of all caches, but this adds to the cost of each processor. [Howe Moxon 1987]

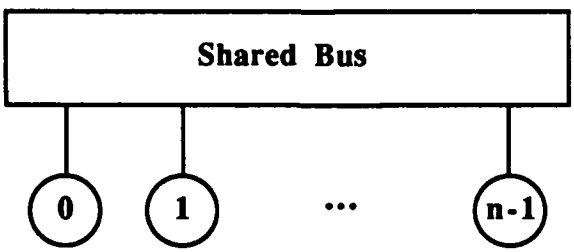


Figure 2-12. A single shared bus.

Crossbar Switch

In the *crossbar switch*, every processor is logically connected to every other. The scheme is implemented by using n^2 switches to connect n processors and n memories, as shown in Figure 2-13. A matrix of interconnection points, called *crosspoints*, connects system elements. All possible distinct connections between the processors and memories are supported, in that by setting the switch appropriately, every PE can access any memory module in a single time unit. [Siegel 1984]

In a crossbar switch, there is never contention for communication resources, although there might be contention for memory. No calculation, other than address translation, is required to establish a route. [Haynes Lau Siewiorek Mizell 1982] Unfortunately, the number of crosspoints required is proportional to the square of the number of processors. This complexity growth tends to eliminate the crossbar as a viable design in all but the simplest configuration option, not only because of the complexity but because of pinout, power, and size considerations. [Welch 1984]

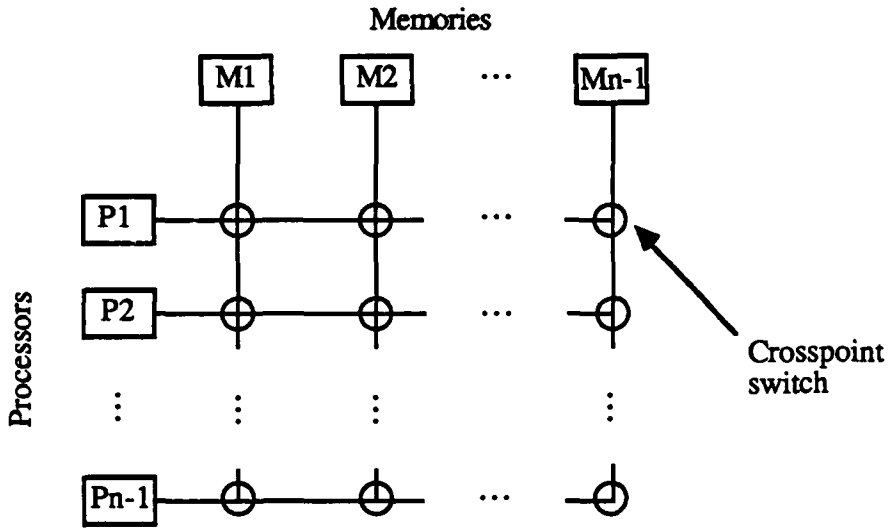


Figure 2-13. A crossbar switch.

Multistage Interconnection Networks (MINs)

The following information on multistage interconnection networks is taken from [Bhuyan 1987], [Howe Moxon 1987] and [Siegel 1984].

A machine based on a *multistage interconnection network* (MIN) connects processors and memory modules through a specialized switching network. The entire memory can be accessed by any processor as in a bus-based system, but MINs can expand to at least 200 processors because the switching network expands and the switching bandwidth increases as processors are added. Many processors can simultaneously access many memories because multiple paths exist through the network.

Formally, an $n \times n$ MIN connects n processors to n memories. For n a power of two, it generally uses $\log_2(n)$ stages of 2×2 switches, each with two inputs and two outputs, with $n/2$

switches per stage. The connection between an input and an output depends on a control bit c provided by the input. When $c = 0$, the input is connected to the upper output; when $c = 1$, it is connected to the lower output.

Many significant MINs have been proposed, including the *omega network*, the *flip network*, the *indirect binary n-cube network*, the *SW-Banyan network*, the *butterfly network*, the *multistage shuffle-exchange network*, the *baseline network*, the *delta network*, and the *generalized cube network*. This is often quite confusing, but one should keep in mind that all of these $\log_2(n)$-stage networks are functionally equivalent and differ only in the interconnection between the adjacent stages. For example, an omega network is characterized by a perfect shuffle interconnection preceding every stage of the switches, as illustrated in Figure 2-14. A butterfly network, illustrated in Figure 2-15, is made up of "butterfly" patterns, hence its name.

Other MINs include the *extra stage cube (ESC) network*, the *F-network*, the *dynamic redundancy network*, the *IADM network*, the *β-network*, the *C-network*, the *INDRA network* [Adams Agrawal Siegel 1987], the *data manipulator network* [Siegel 1979], and the *Benes network* [Feng 1981].

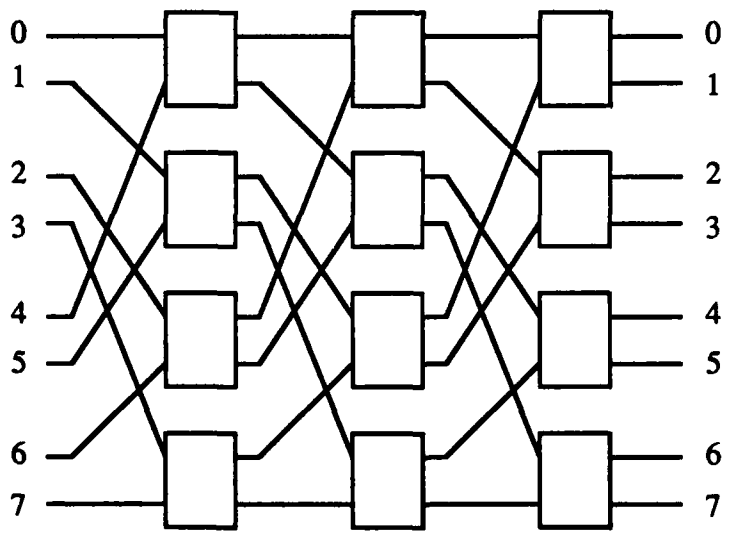


Figure 2-14. An 8 × 8 omega network.

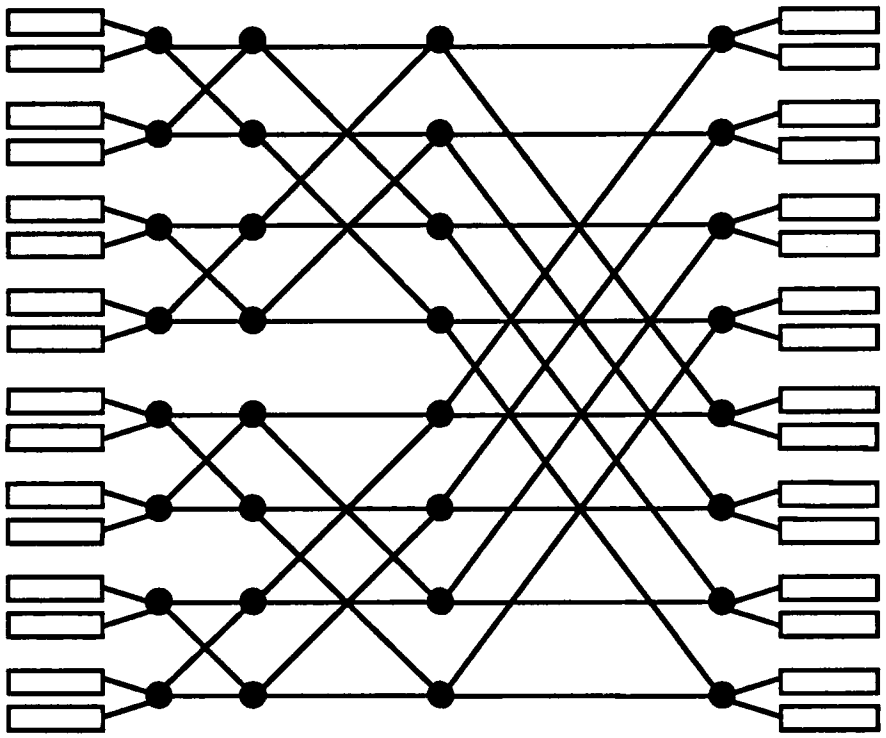


Figure 2-15. A 32-node Butterfly network.

CHAPTER 3

PARALLEL COMPUTERS

Introduction

This chapter gives a brief description of a variety of existing parallel computers, including commercially available machines and research computers. The quantity and quality of information varies from machine to machine due to the availability of literature supplied by the vendors and research centers associated with each machine.

THE CONTENTS OF THIS CHAPTER ARE BASED ON INFORMATION AVAILABLE THROUGH LATE 1987 AND WILL CHANGE WITH THE NEXT EDITION OF THIS DOCUMENT, DUE TO BE RELEASED IN LATE 1988.

3-1 Commercially Available Parallel Computers

Alliant FX/8

The following information is based on [Hwang 1987].

The Alliant FX/8 is a shared-memory multiprocessor system manufactured by Alliant Computer Systems Corporation. The FX/8 system contains up to 8 processing elements with vector capability and up to 12 interactive processors with independent I/O channels. The interactive processors execute interactive user jobs, operating system tasks, and I/O.

Alliant's FX/Fortran compiler automatically identifies opportunities for fine-grained parallel

processing and vector execution and generates globally optimized code that executes efficiently on multiple processors. C and Ada are currently being extended to take maximum advantage of Alliant's parallel architecture. Alliant's Concentrix operating system, a multi-user UNIX 4.2BSD-based operating system, supports fully concurrent parallel processing.

ASPRO (Associative Processor)

The following information is based on [Goodyear 1984], [Law Miller 1983], and [Loral ASPRO 1982].

The ASPRO is an extremely small (0.44 cubic feet) SIMD computer manufactured by Loral Systems Group, a division of Loral Corporation. One particular ASPRO system designed by Loral Systems for usage on the NAVY/Grumman E-2C AEW aircraft consists of 1792 processing elements (PEs) with 8192 bits of local memory per PE. It is possible to configure the ASPRO with a much larger array of PEs and local memory when a size restriction like that of the E-2C is not required.

Figure 3-1 illustrates the block diagram of ASPRO. The Array consists of 14 array groups. Each array group consists of a 128-word by 8192-bit array of solid-state multidimensional access storage and 128 processing elements (128 * 14 = 1792 PEs). The Array Control unit performs all conventional (sequential) data manipulations and drives the Array unit, which performs associative (parallel) operations. The Array Control unit also provides program storage as well as program execution. The Control Memory is made up of three types of storage: (1) buffer memory, (2) program memory, and (3) fixed (or read-only) memory. The Register and Arithmetic subsystem contains one 32-bit and one 16-bit data bus, plus sixteen 16-bit general registers, the array select register, and nine special registers. There is also an arithmetic and logic unit in this subsystem used for both data processing and for memory address generation. The Program and Execution Control Subsystem provides for controlling the sequence of execution of instructions store in the program memory.

ASPRO can run VAX/VMS or UNIX operating systems. Fortran, OPS-83 (an expert systems tool), and ASPRO Assembler programming languages are supported. Ada and C will be available

for the Advanced VHSIC ASPRO. The ASPRO costs approximately $500,000.

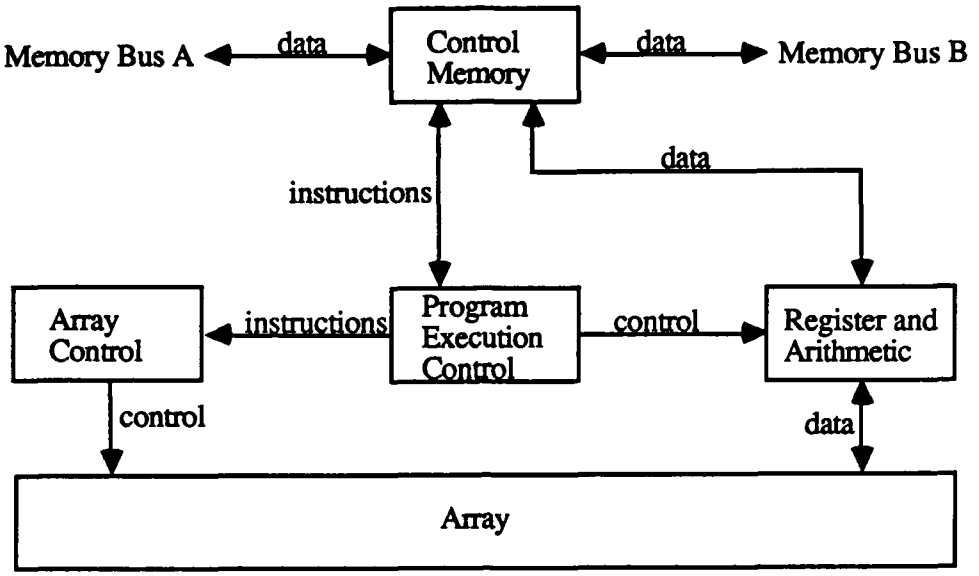


Figure 3-1. Block Diagram of ASPRO.

Butterfly Parallel Processor

The following information is based on [BBN 1986].

The Butterfly Parallel Processor is a tightly coupled shared memory MIMD machine manufactured by Bolt, Berenek and Newman (BBN). (Note that BBN likes to view the Butterfly as a shared memory machine, but in reality the memory is distributed among the nodes.) Each node independently executes its own sequence of instructions, referencing data as specified by the instructions. Nodes are tightly coupled by the *Butterfly Switch*. Tight coupling permits efficient interprocessor communication and allows each processor to access all system memory efficiently.

The Butterfly Parallel Processor consists of processors with memory (referred to as *nodes*) and a log-stage switch interconnecting the processors (the Butterfly Switch). The local memory of the nodes collectively forms the shared memory of the machine. That is, any node can access any of the memory of the machine using the Butterfly Switch. The only difference between references to a node's own memory and references to memory on other node's is that remote references take a little longer. A Butterfly system can be configured with from 1 to 256 nodes, each with 1 to 4 megabytes of memory. Each node is capable of executing 0.5 MIPS.

The Butterfly Switch is illustrated in Figure 3-2. Each Butterfly switching node is a 4 input-4 output switching element. There is a path through the switch network from each node to every other node. Operation of the switch works as follows. The switching nodes use packet address bits to route the packet through the switch from source to destination node. Each processor uses two bits of the packet address to select one of its four output ports. Figure 3-3 illustrates how node #4 sends a message to node #7. Node #4 builds a packet containing the address of node #7 ($7_{10} = 0111_2$) and the message data and sends the packet into the switch. The first switching node strips the two least significant address bits (11) off of the packet and uses them to switch the remainder of the packet out its port 3 ($3_{10} = 11_2$). The next switching node strips off the next two address bits (01) to switch the packet out its port 1 ($1_{10} = 01_2$) to node #7. Notice that the structure of the switch network ensures that packets with address 0111 will be routed with the same number of steps to node #7 regardless of the node sending them.

Application programs run under the Butterfly Chrysalis Operating System, which provides a familiar, UNIX-like environment that supports programming in high-level languages. The Butterfly Parallel Processor is usually programmed in C (extended for use in a parallel processing environment), although Lisp and Fortran (both extended forms) are also available. Programs are written using a cross-compiler and other software development tools on a front-end machine (a DEC VAX or a Sun Workstation, both running 4.2BSD UNIX). The typical development cycle consists of editing, compiling, and linking a program on the UNIX front-end, and downloading, running, and debugging the program on the Butterfly system. A source language debugger for the C language is available that runs on the front-end allowing cross-network debugging of programs running on the Butterfly system.

The Butterfly is one of the Defense Advanced Research Projects Agency's (DARPA's) three Strategic Defense Initiative (SDI) machines. The other two are the WARP at Carnegie Mellon University and the Connection Machine at Thinking Machines Corporation.

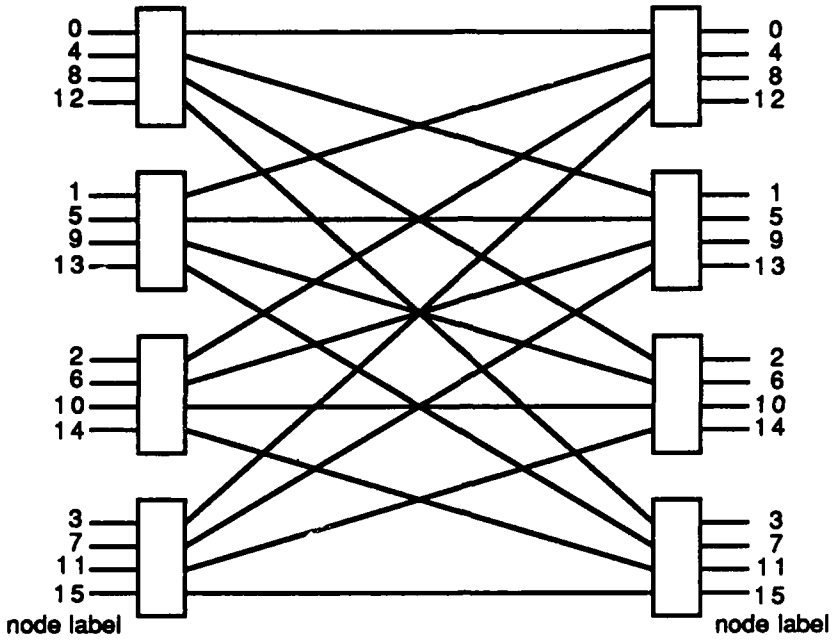


Figure 3-2. A 16 input - 16 output Butterfly Switch.

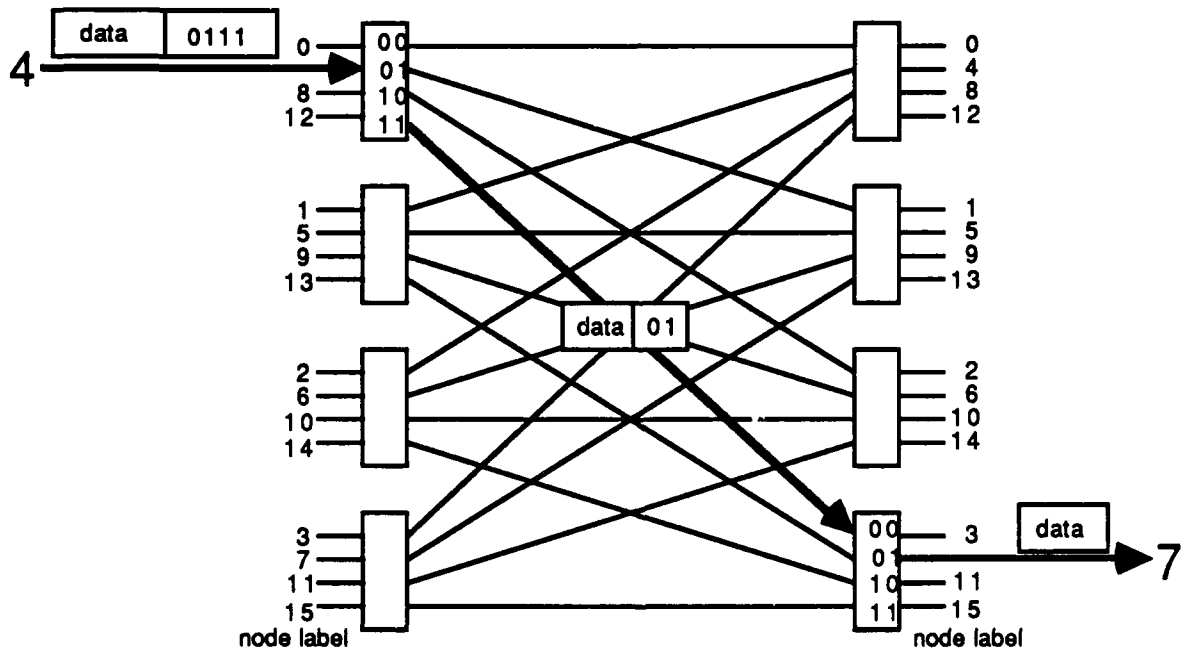


Figure 3-3. A packet in transit through a Butterfly Switch.

Burroughs Scientific Processor (BSP)

The following information on the BSP is based on [Hwang 1987], [Kozdrowicki Theis 1984], and [Kuck Stokes 1984].

The BSP was a commercial attempt made by the Burroughs Corporation beyond the Illiac-IV in

order to meet the increasing demand of large-scale scientific and engineering computers. The BSP is an SIMD machine with 16 processing elements driven synchronously by a single instruction stream. The BSP uses a crossbar network. With a peak speed of 50 MFLOPS, the BSP was designed to perform large-scale computations in the fields of numerical weather prediction, nuclear energy, seismic processing, structure analysis, and econometric modeling. The BSP is not a stand-alone computer - it is a back-end processor attached to a host machine.

Celerity 6000

The following information is based on [Stafford 1987] .

The Celerity 6000 is a vector computer manufactured by Celerity. Celerity describes the 6000 as a system for supercomputing at the department level. It is built upon Celerity's RISC (Reduced Instruction Set Computer) architecture, and it contains a vector processor for processing huge arrays at high speeds. The 6000 can be configured with up to 4 processors and over one gigabyte of memory.

The 6000 supports automatic vectorization of Fortran programs through a preprocessor. The preprocessor takes sequential Fortran77 code and automatically generates code containing directives for the Celerity vector processor. The resulting code is run through the Celerity compiler producing object code which takes full advantage of the vector coprocessor, which has eight 1024-element vector registers.

Connection Machine

The following information on the Connection Machine is based on [Hillis 1987], [Hyde 1987], and [TMC 1987].

The Connection Machine (CM) is a massively parallel processor manufactured by Thinking Machines Corporation. The CM-1 model consists of 16,384 1-bit processing elements (PEs) with 4 Kbits of memory per processor. The CM-2 model can be configured with 65,536 1-bit PEs and 64 Kbits of memory per processor. Two independent interconnection networks are used, a 2-dimensional mesh and a hypercube. That is, groups of 16 PEs are connected in a 4×4 mesh,

and these groups are connected to each other via a hypercube network. A Connection Machine costs between $1 and $6 million.

The programmer interacts with the Connection Machine through a host computer. The processors of the Connection Machine are connected with the host. Programs for the Connection Machine are surprisingly similar to conventional programs. The main difference is that many operations normally carried out by repetitive loops are replaced by single operations corresponding to the simultaneous operation of many processors in the Connection Machine. The routing hardware automatically establishes the necessary communication paths.

The CM-1 must be programmed in PARIS (PARallel Instruction Set), an assembly language, or *Lisp, an extension of Common Lisp. The CM-2 will be able to be programmed in C++ as well. A Fortran 8x compiler is under development.

The Connection Machine is one of the Defense Advanced Research Projects Agency's (DARPA's) three Strategic Defense Initiative (SDI) machines. The other two are the WARP at Carnegie Mellon University and the Butterfly at Bolt, Beranek, and Newman, Incorporated (BBN).

Convex C-1

The following information is based on [Datapro 1987] .

The Convex C-1 is a minisupercomputer manufactured by Convex Computer Corporation. Two models are manufactured, the C-1 XL and the C-1 XP. The C-1 XL is a replacement for the original C-1 with about the same power, but for less money. It has a peak performance of 60 MIPS and 40 MFLOPS and is available in a variety of configurations at an entry-level price of approximately $350,000. It can be configured with up to 64 megabytes of memory. The C-1 XP is a higher performance system featuring up to 1 gigabyte of main memory and is modeled after the Cray X-MP series. A basic C-1 XP system costs approximately $475,000.

The C-1 systems operate under Convex Unix, based on 4.2BSD. Convex provides an automatic vectorizing Fortran compiler as well as a vectorizing C compiler. Plans for a vectorizing Ada compiler are underway.

Up to 240 C-1 XL and C-1 XP processors may be linked together in parallel through an 80

megabit-per-second fiber optic or coaxial cable data path called the Convex Extended Supercomputing Interconnect (CXSI). Convex claims that this interconnection allows for file sharing, job distribution, and load balancing. Each processor in such a multiprocessor configuration retains a complete system with its own CPU, local memory, I/O subsystem, and its own copy of the Convex Unix operating system.

Cray-1

The Cray-1 is a vector supercomputer manufactured by Cray Research, Incorporated. It is classified as a register-to-register vector computer. It has 128 instructions in its instruction set, 4 Kbytes memory per vector register, 32 Mbytes of main memory, and 12 unifunction pipes for vector, scalar, floating point, and fixed point operations. The Cray-1 has a peak speed of 160 MFLOPS. It is programmed using Cray Fortran (CFT) with automatic vectorization. Compatible front-end host machines include IBM, CDC, and Univac mainframes. [Tutorial 1984]

Cray-2 and Cray-2S

The following information is based on [Cray 1987].

The Cray-2 and Cray-2S series of computer systems are vector supercomputers manufactured by Cray Research, Incorporated. The various models are as follows:

Model	# Processors	Memory Size (million 64-bit words)	Cost (million)
Cray-2/4-256	4	256	$15.5
Cray-2/4-128	4	128	$14.5
Cray-2/2-128	2	128	$12
Cray-2S/4-128	4	128	$17.5
Cray-2S/2-128	2	128	$15.5
Cray-2S/2-64	2	64	$12

These vector supercomputers have a peak speed in the range of 120 MFLOPS to 2 GFLOPS. Cray Research provides two automatic vectorizing Fortran compilers: CFT2 (Cray Fortran Compiler

version 2) and CFT77 (Cray Fortran compiler that fully compiles with the ANSI 1978 standard). In addition, Cray is developing an automatic vectorizing C compiler and a Pascal compiler which automatically vectorizes for loops.

The Cray-2 and Cray-2S machines run under the UNIX Cray Operating System (UNICOS), based on AT&T's UNIX System V operating system. The main difference between the Cray-2 and Cray-2S is that the Cray-2 uses DRAM (dynamic RAM) common memory while the Cray-2S uses faster SRAM (Static RAM) common memory. The Cray-2S also has faster raw chip speed and reduced memory contention.

Cray supercomputers can operate under two modes of operation. Multitasking allows two or more parts of a program (tasks) to be executed in parallel sharing a common memory space. Multiple processors may also operate independently and simultaneously on separate jobs for greater system throughput or may be applied in any combination to operate jointly on a single job for better program turnaround time.

Cray X-MP

The following information is based on [Cray 1987].

The Cray X-MP series of computer systems are vector supercomputers manufactured by Cray Research, Incorporated. The various models are as follows:

Model	# Processors	Memory Size (million 64-bit words)	Cost (million)
Cray X-MP/416	4	16	$16
Cray X-MP/48	4	8	$14
Cray X-MP/44	4	4	$12
Cray X-MP/216	2	16	$10.5
Cray X-MP/28	2	8	$9
Cray X-MP/24	2	4	$7.5
Cray X-MP/22	2	2	$6
Cray X-MP/116	1	16	$8.5
Cray X-MP/18	1	8	$7
Cray X-MP/14	1	4	$5.5
Cray X-MP/14SE	1	4	$2.5

To summarize, the Cray X-MP/4 can be configured with 4, 8, or 16 million 64-bit words of shared memory, the Cray X-MP/2 can be configured with 2, 4, 8, or 16 million 64-bit words of shared memory, the Cray X-MP/1 is a single high-performance CPU which can be configured with 4, 8, or 16 million 64-bit words of shared memory, and the Cray X-MP/14SE combines a single Cray X-MP CPU with 4 million 64-bit words of static MOS memory (specially packaged and priced to serve both first-time supercomputer users and dedicated project requirements in large-scale computational environments). As an example, the Cray X-MP/4 has a peak speed of 840 MFLOPS.

Cray Research provides two automatic vectorizing Fortran compilers for the X-MP series: CFT (Cray Fortran Compiler) and CFT77. Both compilers fully meet the ANSI 78 standard. In addition, Cray also offers a vectorizing C compiler and a vectorizing ISO level 1 Pascal compiler. A variety of operating systems are available. COS (Cray Operating System) manages high-speed data transfers between the Cray X-MP and peripherals. UNICOS (UNIX COS), based on AT&T's UNIX System V operating system, is available as a stand-alone operating system or as a guest operating system running concurrently with COS. The Cray TimeSharing System (CTSS) is also available (see the section on the SCS-40 for more information on CTSS).

Cray supercomputers can operate under two modes of operation. Multitasking allows two or more parts of a program (tasks) to be executed in parallel sharing a common memory space. Multiple processors may also operate independently and simultaneously on separate jobs for greater system throughput or may be applied in any combination to operate jointly on a single job for better program turnaround time.

Cyber 205

The Cyber 205 is a vector supercomputer manufactured by Control Data Corporation. It is classified as a memory-to-memory architecture and has 32 megabytes of main memory and up to 4 vector processors with 6 pipes in each separate string unit and scalar unit. The Cyber 205 has a peak speed of 800 MFLOPS. It is programmed using Cyber 200 Fortran with automatic vectorization. [Tutorial 1984]

Cyberplus

The following information is based on [Hwang 1987].

The Cyberplus is a distributed memory, MIMD computer manufactured by Control Data Corporation. The Cyberplus uses a multi-ring structure for packet-switched message passing among processors (see Figure 3-4). The configuration consists of a maximum of four ring groups with 16 Cyberplus processors (accelerators) per group. Each group has two rings: the 16-bit system ring provides communications between the Cyberplus processors and the host Cyber processor, and the 16-bit application ring provides direct communications among the Cyberplus processors themselves. This ring structure can carry $2n$ data packets simultaneously in the ring which links n processors. Besides the dual rings, an additional memory ring can be added to provide direct memory-to-memory communication among Cyberplus processors and between a Cyber processor 64-bit memory and a Cyberplus processor 64-bit memory.

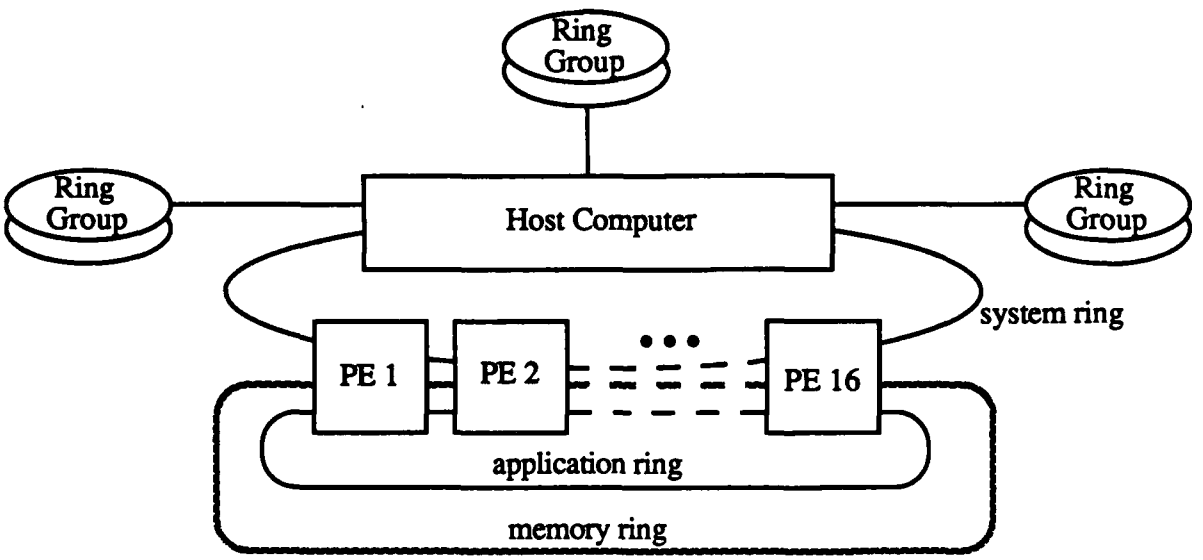


Figure 3-4. Interprocessor communication structure in the CDC Cyberplus parallel processing system.

Distributed Array Processor (DAP)

The following information on the DAP is based on [Baillie 1986], [Hwang Briggs 1984] and [Uhr 1984].

The DAP was designed by International Computer Limited (ICL) in England. The DAP is an SIMD mesh computer which can be constructed in groups of 16 1-bit processing elements (PEs) in

various sizes, such as 32 × 32, 64 × 64, 128 × 128, and 256 × 256. Each PE is linked to its four nearest neighbors: north, south, east and west. Each PE has 4000 bits of local memory. The DAP was developed for numerical problem solving.

Programs for the DAP consist of two parts: a serial part written in Fortran 77 which executes on the host, and a parallel part written in a matrix and vector extension of Fortran IV called DAP Fortran. The two parts of the program communicate through Fortran common blocks.

ELXSI System 6400

The System 6400 is a tightly-coupled, bus-oriented, shared-memory MIMD computer manufactured by ELXSI, a subsidiary of Trilogy, Limited. It can be configured with up to 12 processing elements. The System 6400 employs a message based operating system called Embos. A number of programming languages are supported including Fortran, C, Pascal, and COBOL. The cost of a fully configured 12-processor system is approximately $3 million. [Frenkel 1986] [Hays 1986] [Olson 1985]

Encore Multimax

The Encore Multimax is a shared memory, MIMD computer manufactured by Encore Computer Corporation. The Multimax can be configured with 2 to 20 processors and from 4 to 128 megabytes of memory. Processors are connected by a 100 Mbytes/second Nanobus, so named because it is one foot long - approximately the distance traveled by light in one nanosecond. The Multimax's operating system is UMAX, a UNIX-based operating system. Programming languages C and Fortran 77 are provided. Encore claims that Parallel Ada will be available soon. The Multimax costs under $1 million. [Encore 1987] [Moore Nassi O'Neil Siewiorek 1986]

ETA^{10}

The following information is based on [Emmen 1987], [ETA 1987] and [Hwang 1987].

The ETA^{10} is a shared-memory multiprocessor supercomputer extended from the Cyber 205. It is manufactured by ETA Systems, Inc., a subsidiary of Control Data Corporation. Figure 3-5

shows the system components of the ETA^{10}. The ETA^{10} can have up to 8 processing elements (PEs), each with 32 Mbyte of memory, and up to 18 I/O processors under the coordination of a service processor. The shared memory can hold from 64 upto 2,000 Mbyte. The communication buffer is used for fast transfer of information among central processors and I/O units. The ETA^{10} has a peak performance of 10 billion calculations per second. Prices vary from $996,000 for the slowest model (ETA^{10}-P) to $8.9 million for the fastest model (ETA^{10}-G).

The ETA System V operating system is UNIX compatible. The C compiler is non-vectorizing, but the Fortran compiler does include automatic vectorization and supports Fortran 8x notation.

A simulator, called the ETA^{10} Multiprocessing Simulator, has been developed for the ETA^{10} on the CDC Cyber 205. This tool executes applications written in Fortran that are structured for multiprocessing. The Simulator introduces structures for parallel processing that match the architecture of the ETA^{10} system. Users can experiment with ideas of parallel processing, including manipulating the global shared memory, synchronizing on semaphores, and moving data between the private memories of the individual processors. The Simulator generates a history that tracks all changes of state for tasks, semaphores, and processors.

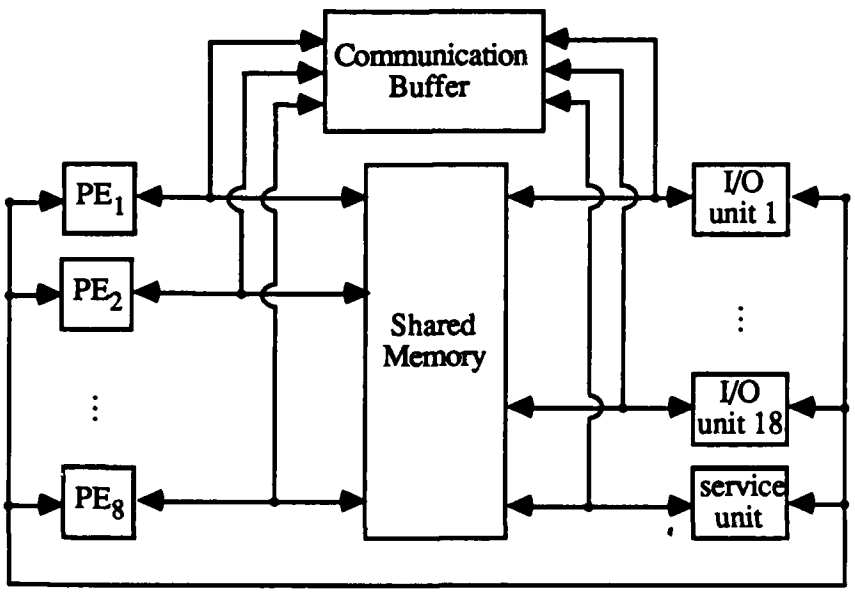


Figure 3-5. ETA^{10} system components.

FACOM Vector Processor Systems

The following information is based on [Miura Uchida 1984].

The FACOM Vector Processor System is a vector supercomputer manufactured by Fujitsu Limited in Kawasaki, Japan. Two models are manufactured, the VP-100 and the VP-200. These machines employ multiple pipeline units which can operate concurrently. Peak speeds of the VP-100 and VP-200 are 267 MFLOPS and 533 MFLOPS, respectively.

The FACOM Vector Processor System (Figure 3-6) consists of a scalar unit, a vector unit, and a main storage unit. The *main storage unit* has a maximum capacity of 256 megabytes for the VP-200 and 128 megabytes for the VP-100. The *scalar unit* fetches and decodes all instructions. There are 277 instructions, 195 of which are scalar instructions and 82 of which are vector instructions. Scalar instructions are executed in the scalar unit, while vector instructions are issued to the vector unit.

The *vector unit* mainly consists of six functional pipeline units, vector registers, and mask registers. The functional pipeline units are: add/logical pipe, multiply pipe, divide pipe, mask pipe, and two load/store pipes. The first three are used for arithmetic operations and any two can operate concurrently.

Fujitsu has developed the Fortran77/VP vectorizing compiler for the FACOM Vector Processor System. This compiler vectorizes not only the simple do loops but also vectorizes nested do loops. It also detects and separates recurrences.

FLEX/32

The FLEX/32 is an MIMD shared memory multiprocessor manufactured by Flexible Corporation. The FLEX/32 is configured with 20 processors. The machine has ten local buses, each of which connects two processors. These local buses are connected together and to shared memory by a common bus. [Fatoohi Grosch 1987]

GEC Rectangular Image and Data processor (GRID)

The GRID is an SIMD computer with 4096 (64 × 64 array) bit-serial processing elements (PEs),

each with 8 Kbits of local memory and connections to all eight neighboring PEs (Figure 3-7). In addition, GRID contains a scalar processor to deal with serial code. GRID is programmed in a parallel extension of C called GRID-extended C. GRID is hosted by a minicomputer running the UNIX operating system. [Baillie 1986] [Tucker 1986]

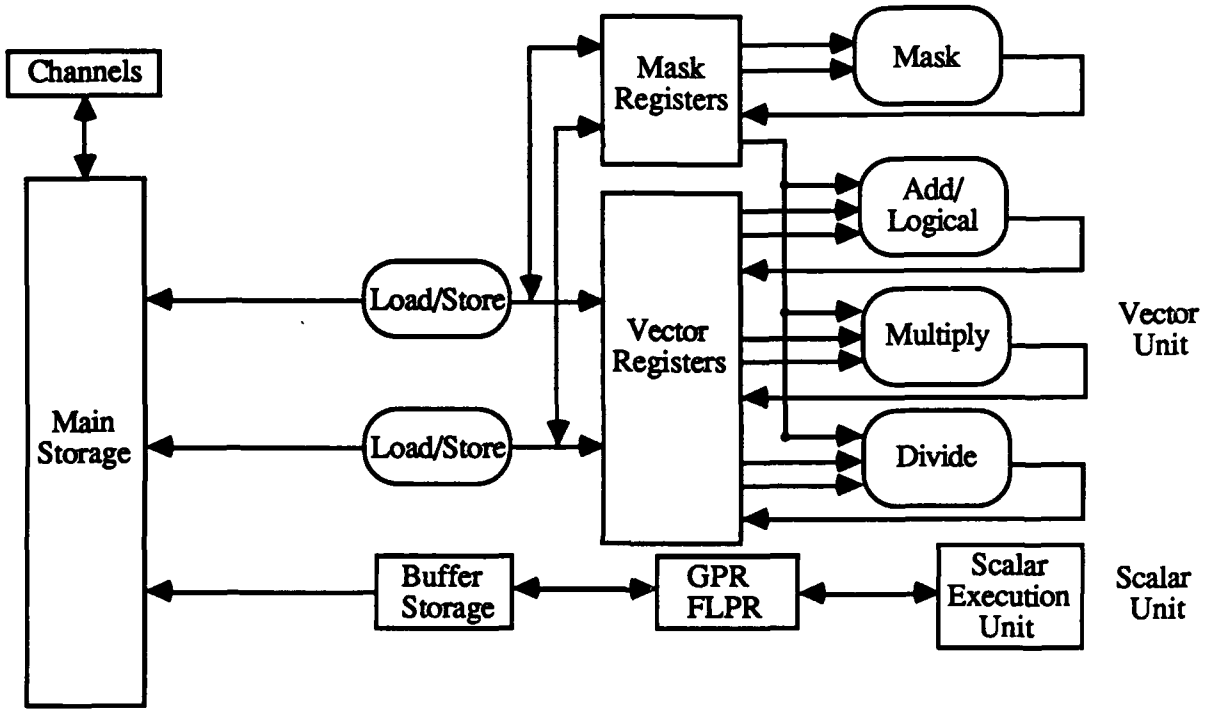


Figure 3-6. FACOM Vector Processor System Model Diagram

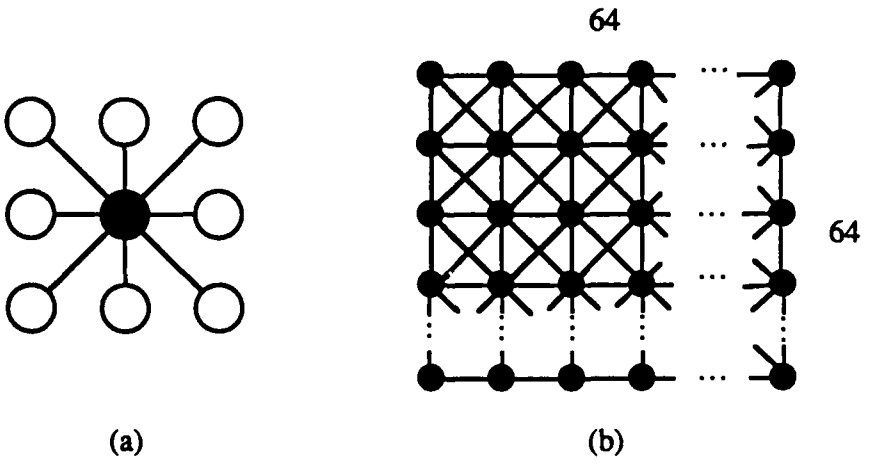


Figure 3-7. (a) 8-connectivity (b) GRID interconnection network

Heterogeneous Element Processor (HEP)

The following information is based on [Frenkel 1986], [Hwang 1987] and [Smith 1984].

The HEP computer system is an MIMD computer that was manufactured by Denelcor, Inc. until Denelcor went bankrupt in 1986. (Only six $7 million HEP machines were sold.) The processing elements (PEs) in the HEP were pipelined to support many concurrent processes, with each pipeline segment responsible for a different phase of instruction interpretation. Each processor had its own program memory, general purpose registers, and functional units. A number of these processors were connected to shared data memory modules by means of a very high speed pipelined packet switching network. The HEP was programmed in HEP Fortran, an extension to Fortran which allowed the programmer to write explicit parallel algorithms.

The HEP-1 model consisted of 16 processors and up to 128 memory modules that were connected via a pipelined packet switching network. Parallelism was exploited at the process level within each processor. The system allowed 50 user processes to be concurrently created in each processor. Fifty instruction streams were allowed per processor, with a maximum of 50 * 16 = 800 user instruction streams in the entire HEP system.

IBM GF11

The following information is based on [Beetem Denneau Weingarten 1985], [Foulser Schreiber 1987] and [Hwang 1987].

GF11 is a parallel computer currently under construction at the IBM T.J. Watson Research Center. GF11 is a modified SIMD computer designed specifically for the numerical solutions of problems in quantum chromodynamics. Peak speed for the GF11 is 11 GFLOPS.

Figure 3-8 illustrates the block diagram for GF11. The machine consists of 576 floating-point processors (512 + 64 spare processors to be enabled if a primary processor fails). The processors are interconnected by a three stage full Benes network called the *Memphis switch*. This switch is a non-blocking switch capable of realizing configurations incorporating any permutation of the processors and instantaneous reconfiguration. Each stage of the Memphis switch consists of 24 24-input crossbar switches. The middle stage is connected to the outer stages by perfect shuffle

fixed interconnections. By suitable configurations of the crossbars it is possible to realize any permutation of the 576 inputs. For example, this switch allows the GF11 to be organized into any of a number of different topologies, such as a rectangular mesh of any dimension and size, any torus, a hexagonal mesh, or some irregular organization matching perfectly with a special problem. The Central Controller has several functions: (1) storage and broadcast of GF11 instruction streams, (2) address relocation and remap, (3) communication with the Host CPU, and (4) status and error checking.

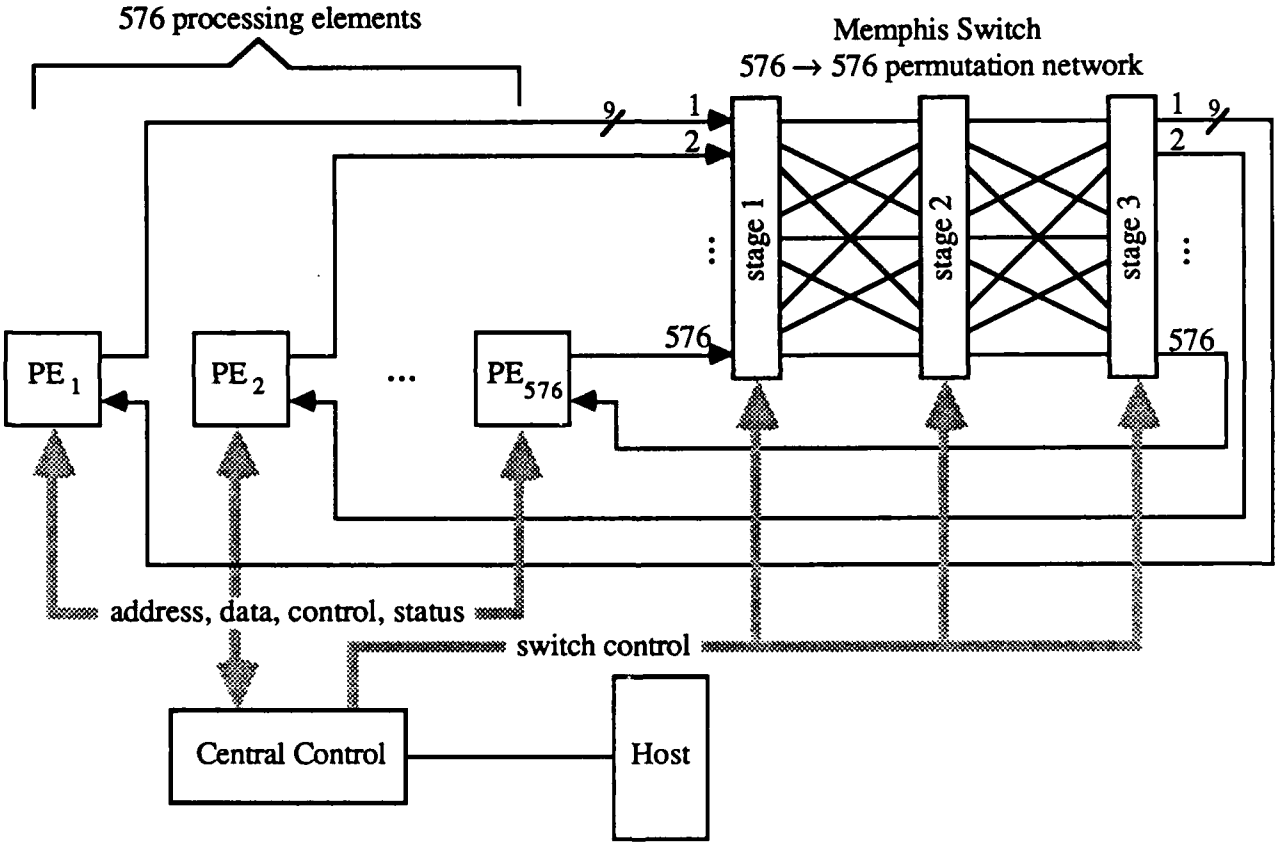


Figure 3-8. The IBM GF11.

Illiac IV

The following information is based on [Welch 1984].

The Illiac IV is a 64 processor SIMD, mesh-connected computer (the processing elements are arranged as an 8 × 8 array). The Illiac IV was designed specifically for integrating partial differential equations encountered in problems such as numerical weather forecasting, fluid

dynamics, and nuclear effects data processing, where a high degree of connectivity between neighboring processors is required in order to efficiently process a finite difference lattice. A single *control unit* (CU) controls the 64 processing elements (PEs) and breaks down the instructions to the point where a microcode sequence is generated to execute the instruction in the PEs. Two kinds of instructions exist and are executed separately and independently, those executed in the CU, and those executed in the PEs. The PE microcode sequence is bussed to all PEs simultaneously.

Figure 3-9 gives a block diagram of the Illiac IV architecture and the control unit. Each PE is basically a four register arithmetic unit with an A register and a B register to hold operands, an S register for temporary storage, and an R register used to transfer information among the PEs in the routing operation. The 64-bit R register of every PE is wired to the R register of four other PEs so that PE_i connects to PE_{i+1}, PE_{i-1}, PE_{i+8}, and PE_{i-8}. The routing operation acts as if the 64 R registers were a 4096 register with an end around shift capability. A route 1 right command causes every R register to be shifted 64 bits to the right. The connection to PE_{i+8} allows rapid movement of data over a longer distance.

Intel iPSC Hypercube System

The following information is based on [Intel 1987].

The iPSC (Intel Personal SuperComputer) is an MIMD, distributed memory, message passing hypercube computer manufactured by Intel Scientific Computers. The iPSC consists of two major functional elements, as shown in Figure 3-10: the *cube* and the *host* (or cube manager). The cube can be configured with 32, 64, or 128 nodes, based on the Intel 80286 chip. Nodes can be configured with either 0.5 megabytes or 4.5 megabytes of local memory. The nodes are connected by high-speed communication channels in a hypercube topology. The host is a microcomputer which is linked to each node over a global Ethernet communication channel.

Each node has a copy of the node operating system. The node operating system provides the application programmer with the necessary set of software services for dynamically loading programs, managing multiple processes, and delivering variable length messages between processes. Languages offered on the iPSC include C, Fortran and Lisp.

Intel also manufactures the iPSC2, the SugarCube and the VX system. The Intel iPSC2 hypercube system allows for each node to have up to 16 megabytes of local memory and is based on the Intel 80386 chip. In addition, node↔node communications and host↔node communications have been improved in the iPSC2. The Intel SugarCube system is a low-cost eight-processor entry level hypercube. The iPSC-VX system is a large-scale parallel computer manufactured by Intel, which can be configured with up to 64 vector processing nodes with a combined peak performance of 424 MFLOPS.

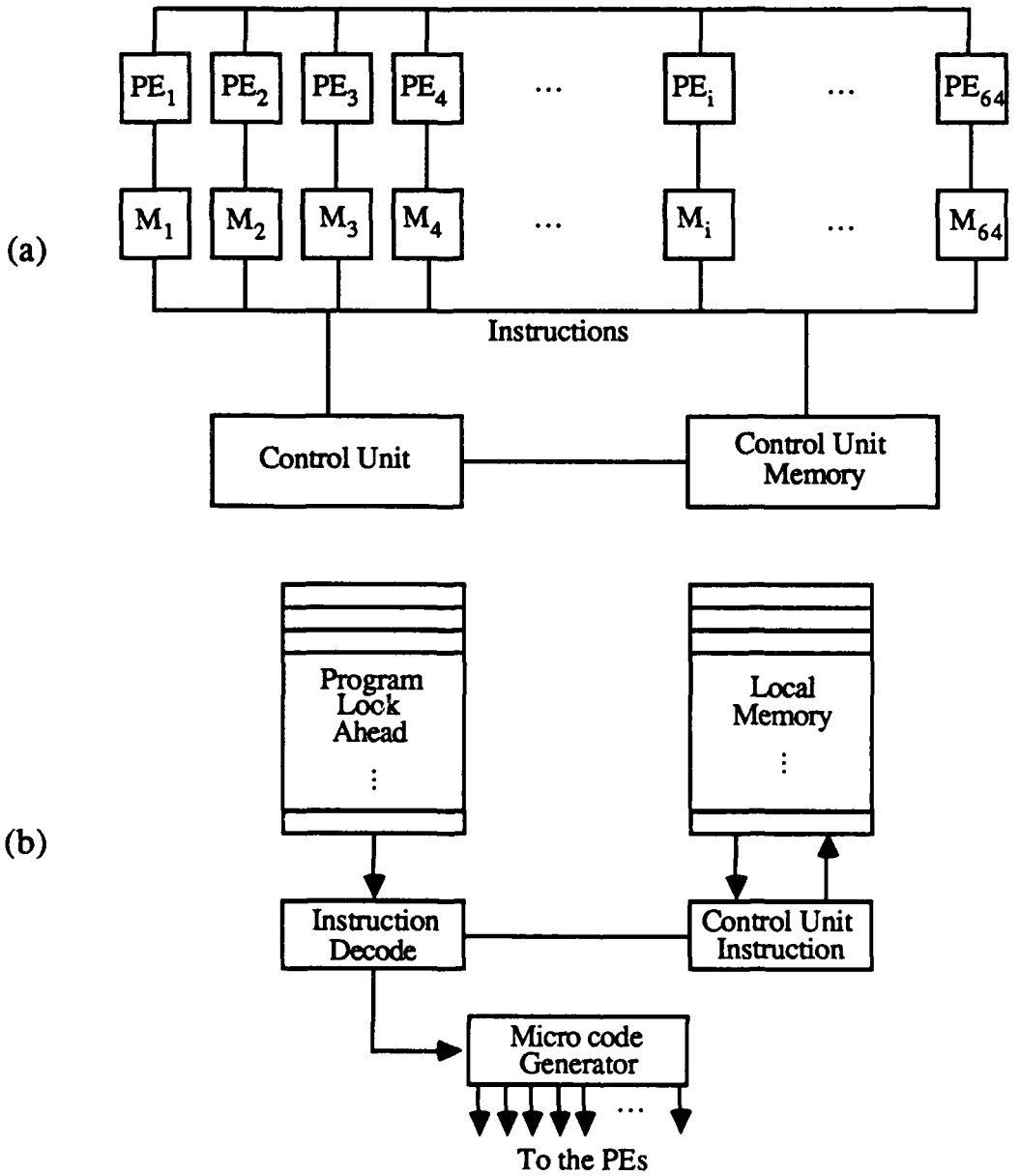


Figure 3-9. Block diagrams of (a) the Illiac IV and (b) the Illiac IV control unit.

IP-1

The IP-1 is a parallel processing minicomputer manufactured by International Parallel Machines, Incorporated. The IP-1 consists of 9 processors (1 master processor and 8 slave processors) and uses a crossbar network. The IP-1 runs the Runix operating system, an operating system which International Parallel Machines claims is the only known Unix look-alike that supports true parallel processing. [IP-1 1987]

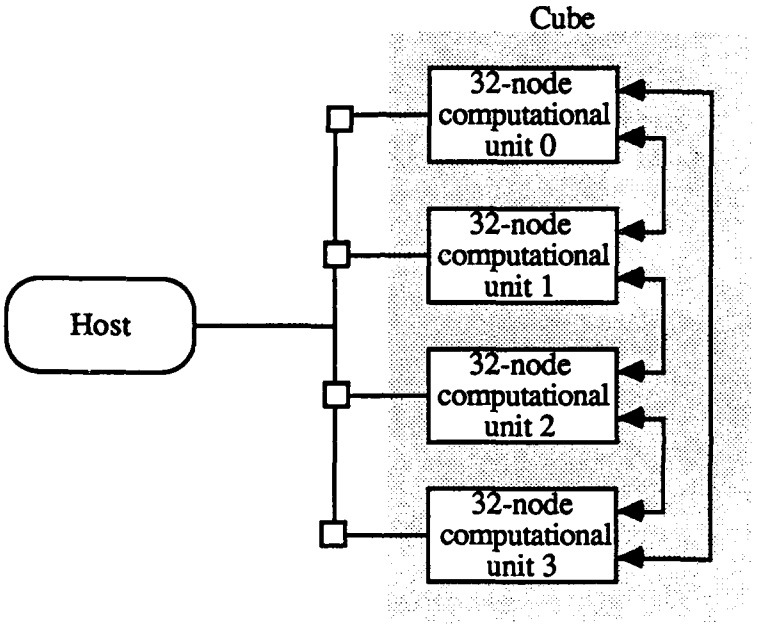


Figure 3-10. The iPSC hypercube can be configured with 32, 64, or 128 nodes.

Massively Parallel Processor (MPP)

The following information is based on [Batcher 1984], [Gilmore 1986], [Loral MPP 1983], and [Palumbo 1987].

The Massively Parallel Processor (MPP) is a distributed memory, message passing, SIMD computer with 16,384 1-bit processing elements (PEs) arranged as a 128 × 128 mesh (square array). Each PE in the 128 × 128 mesh communicates with its four nearest neighbors: north, south, east, and west.

Since the MPP is a 2-dimensional mesh connected computer, the four edges of the processing array must be specially handled. The MPP provides a software configurable method for changing the topology of these edge PEs. For the top and bottom row of PEs, two configurations are

possible. The elements in the top row and the respective elements in the bottom row can either be (1) not connected, or (2) connected. The left and right edge of the processing array can have four configurations. They can be either (1) not connected, (2) left and right elements in the same row connected (cylindrical), (3) left elements connected to right elements in the previous row (linear chain), or (4) left elements connected to right elements in the previous row and last element in last row connected to first element in first row (linear loop).

The major components of the MPP, as shown in Figure 3-11, are the array unit, the array control unit, the program and data management unit, the staging memories, and the host computer. The *array unit* processes arrays of data at high speed and is controlled by the *array control unit*, which also performs scalar arithmetic. Users control the MPP from terminals on a front-end computer (host). The *program and data management unit* controls the overall flow of data and programs through the system and handles certain ancillary tasks such as program development and diagnostics. The *staging memories* buffer and reorder data between the array unit, program and data management unit, and host computer.

The MPP is designed to process satellite imagery at high rates. It has a peak speed of 6.5 billion 8-bit integer additions per second and 470 million 32-bit floating point additions per second. Both MPP-Pascal (Parallel Pascal designed by Anthony Reeves) and Parallel Fortran-77 programming languages are provided. The MPP costs between $2 and $4 million and is manufactured by Loral Systems Group, a division of the Loral Corporation.

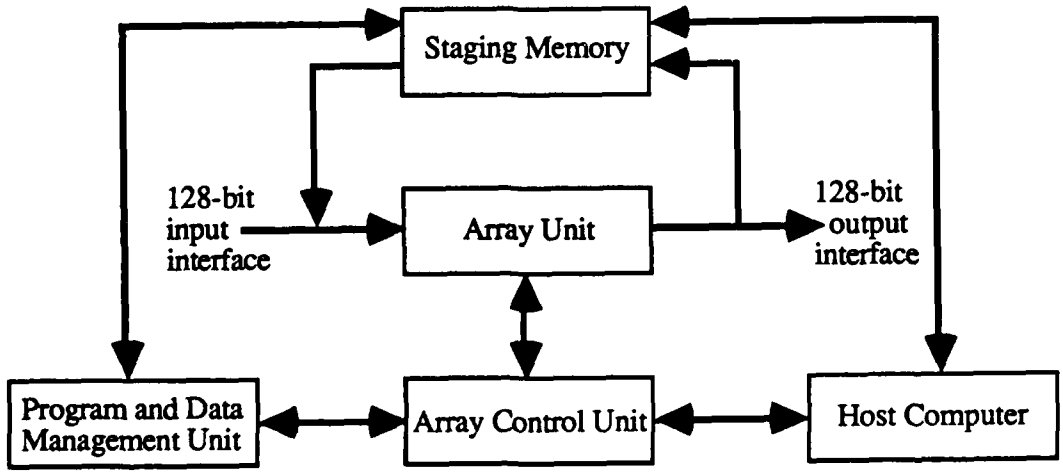


Figure 3-11. MPP System Block Diagram

NCUBE/ten

The following information is based on [NCUBE 1987].

The NCUBE/ten is a family of MIMD, distributed memory, message passing hypercube computers manufactured by NCUBE Corporation. The 'ten' stands for 10-cube; that is, there are $2^{10} = 1024$ nodes in the largest configuration (see *hypercube network* in Section 2-1). The smallest configuration has 16 processors. Cost for an NCUBE hypercube ranges from $10,000 to $1.5 million.

Extended versions of Fortran/77, C and assembly language are available for programming the NCUBE hypercube. These languages are extended with communication facilities for parallel processing. The host computer runs a UNIX-like operating system called AXIS. AXIS can allocate subsets of the hypercube to multiple users. It provides for loading, running, communicating with and debugging programs in the hypercube nodes. A simple operating system nucleus called Vertex runs on each of the nodes. Vertex automatically routes messages through optimal paths to their destination.

NCUBE Corporation also manufactures the NCUBE/four family of parallel computers and the NCUBE/seven family of parallel computers. The NCUBE/four is an entry-level system with 4 processors. The NCUBE/seven is designed for the office environment. It is compact and can be configured with 16 to 128 processors.

SCS-40

The following information is based on [Anderson Grimes Riebman Simon 1987], [McClain 1987].

The SCS-40 is a Cray compatible system designed by Scientific Computer Systems Corporation (SCS). At $60,000, the SCS-40 is far less expensive than the Cray X-MP, the particular Cray it was modeled after, and it performs at 24% to 30% (44 MFLOPS) of the X-MP's speed. The SCS-40 supports concurrent scalar and vector processing.

The operating system for the SCS-40 is the Cray TimeSharing System (CTSS). CTSS is a UNIX-like interactive operating system, augmented by supercomputer-oriented features. It is a modular operating system centered on a small set of kernel services which control the system

resources. CTSS addresses many supercomputer-related issues that conventional operating systems do not. First, CTSS makes efficient use of system resources, and provides tools for better user productivity. For example, CTSS' resource management facilities are explicitly designed for efficient use by multiple users. These facilities allow the user to control the priorities of his/her programs, and the system manager to track and allocate resources. Second, CTSS takes care of file security for the user. Files are, by default, accessible only to the creator of the files. File protections may only be changed by the creator. Third, CTSS understands parallel processing and vector operations, and operates at a very small overhead percentage rate.

SCS provides its own version of the Fortran programming language. SCS Fortran is a full ANSI-77 standard compiler with vectorization capabilities and optimized run-time libraries. It generates very efficient code using vectorization techniques designed to make use of the vector facilities of the SCS-40.

Sequent Balance and Symmetry Series

The following information is based on [Sequent 1986] and [Sequent 1987].

The Balance and Symmetry Series of parallel computer systems is manufactured by Sequent Computer Systems, Incorporated. These systems are configurable general-purpose, shared-memory, MIMD computers that support simultaneous execution of parallel programs and existing sequential applications. Two models are manufactured in each Series: B8 and B21 in the Balance Series and S27 and S81 in the Symmetry Series. Model B8 can be configured with 2 to 12 processors, model B21 can be configured with 2 to 30 processors, and models S27 and S81 can each be configured with 2 to 30 processors. Processors are connected on a global, synchronous system bus. The Balance Series has a peak speed of 21 MIPS for the 30 processor model, and the Symmetry Series has a peak speed of 80 MIPS.

Each system runs the DYNIX operating system. DYNIX is an enhanced UNIX operating system which supports 4.2BSD and System V applications simultaneously. Programming languages C, Fortran, Pascal and Ada are supported. Sequent also provides a parallel version of the UNIX dbx debugger, called Pdbx. Pdbx is specifically designed for debugging multiprocess

parallel applications.

System 14

The following information is based on [Ametek 1987] and [Tucker 1986].

The System 14 is a distributed memory, message passing MIMD hypercube computer manufactured by Ametek, Incorporated. It can be configured with between 16 and 256 processors (groupings of 16-node modules), each with up to 256 megabytes of local memory. The System 14 can achieve a peak speed of 12 - 200 MIPS and 0.8 - 12 MFLOPS (16 - 256 nodes). Compatible host computers include the DEC VAX and MicroVAX II computers. Ametek provides the C programming language. The Ametek XOS operating system provides for task creation and termination within the network, coordination and control of internode message passing, memory buffer management, and error detection and notification.

Ametek provides a number of software tools to simplify programming and to enable debugging of programs on the host machine. A simulator allows program development on the host computer and frees the System 14 for actual computation. It provides full support for operating system calls, user-selectable debugging features, and playback of a simulator run to reproduce an error. It runs in a completely asynchronous mode to model true performance of a program on physical hardware.

A debugger, called Mpdbx, provides multi-process debugging capabilities in simulator runs. It offers the same breakpoint, trace, and debugging features for concurrent programs that Unix source code level debugger dbx provides for serial computers.

User interfaces allow the programmer to simultaneously develop programs for both the simulator and the System 14 hardware without having to maintain two independent sets of files. Two modes are available to the user: *simulator mode* and *hardware* (node) *mode*. This capability provides a directory structure and commands which enable the user to easily change between these two modes and to compile and link programs with the same commands for the simulator and the hardware.

SX-2 Series

The following information is based on [Supercomputing 1987a].

The SX-2 Series are vector supercomputers manufactured by Honeywell-NEC Supercomputers, Incorporated. The SX-2 supports both vector and scalar processing and has a single processor peak performance rate of 1.3 GFLOPS. A RISC-like architecture is used to speed up performance of the scalar unit in the arithmetic processor for those parts of a scientific program that are not vectorizable. The SX-2 has a shared memory with a maximum capacity of 256 Mbytes. The SX-2 is classified as a register-to-register architecture and has 4 sets of vector pipes, each with 4 vector arithmetic units enabling a maximum of 16 parallel vector operations to be performed. The SX family consists of the SX-2-100 (formerly the SX-1E), the SX-2-200 (formerly the SX-1) and the SX-2-400 (formerly the SX-2).

The Fortran 77/SX compiler is an automatic vectorizing compiler with some features not offered on other supercomputers. For example, Fortran 77/SX can handle do loops containing if statements and loops containing vector subscripts.

T Series

The following information is based on [FPS 1986], [FPS 1987] and [Frenkel 1986].

The T Series is a highly parallel scientific computer manufactured by Floating Point Systems, Incorporated. It consists of multiple processors linked together as a hypercube, where each processor has multiple communication channels, a control processor, and a powerful vector processing unit. The highest configuration of the T Series is the T/40000 which has a maximum of 16,384 processing elements.

T Series programs are written in Fortran 77, C, or Occam (see Section 4-6). The T Series uses a standard micro VAX Ultrix operating system.

3-2 Research Computers

Cedar System

The following information is based on [Hwang 1987] and [Kuck Davidson Lawrie Sameh 1986].

Cedar is a research parallel processor being developed at the Center for Supercomputing Research and Development at the University of Illinois, Urbana-Champaign. As shown in Figure 3-12a, Cedar consists of multiple clusters with a globally shared memory. Each cluster is a slightly modified Alliant FX/8 minisupercomputer with a UNIX operating system, eight floating-point processors, fast interprocessor synchronization and vector instructions. (Figure 3-12b illustrates a Cedar cluster.) The Cedar system employs a shuffle network based on 8×8 crossbar switches. The main objective of the Cedar project is to demonstrate that parallel processing can deliver high performance across a wide range of applications.

The operating system being developed for Cedar is called Xylem [Emrath 1985] and is based on the 4.2 version of AT&T Bell Laboratories' UNIX operating system. One of the main purposes of Xylem is to provide multiprocessing primitives needed by the Cedar compiler. Cedar Fortran is a superset of Fortran 8x, with extensions that provide access to specific features of the Cedar hardware and operating system. A translator is available that accepts Fortran77 as input and produces Cedar Fortran as its output.

Cellular Logic Image Processor (CLIP4)

CLIP4 is an SIMD computer designed and built at University College London. It consists of a 96×96 array of 1-bit processors. Each processor is linked to either the six or eight nearest neighbors. CLIP4 was developed for image processing applications. [Tucker 1986] [Tutorial 1984] [Uhr 1984]

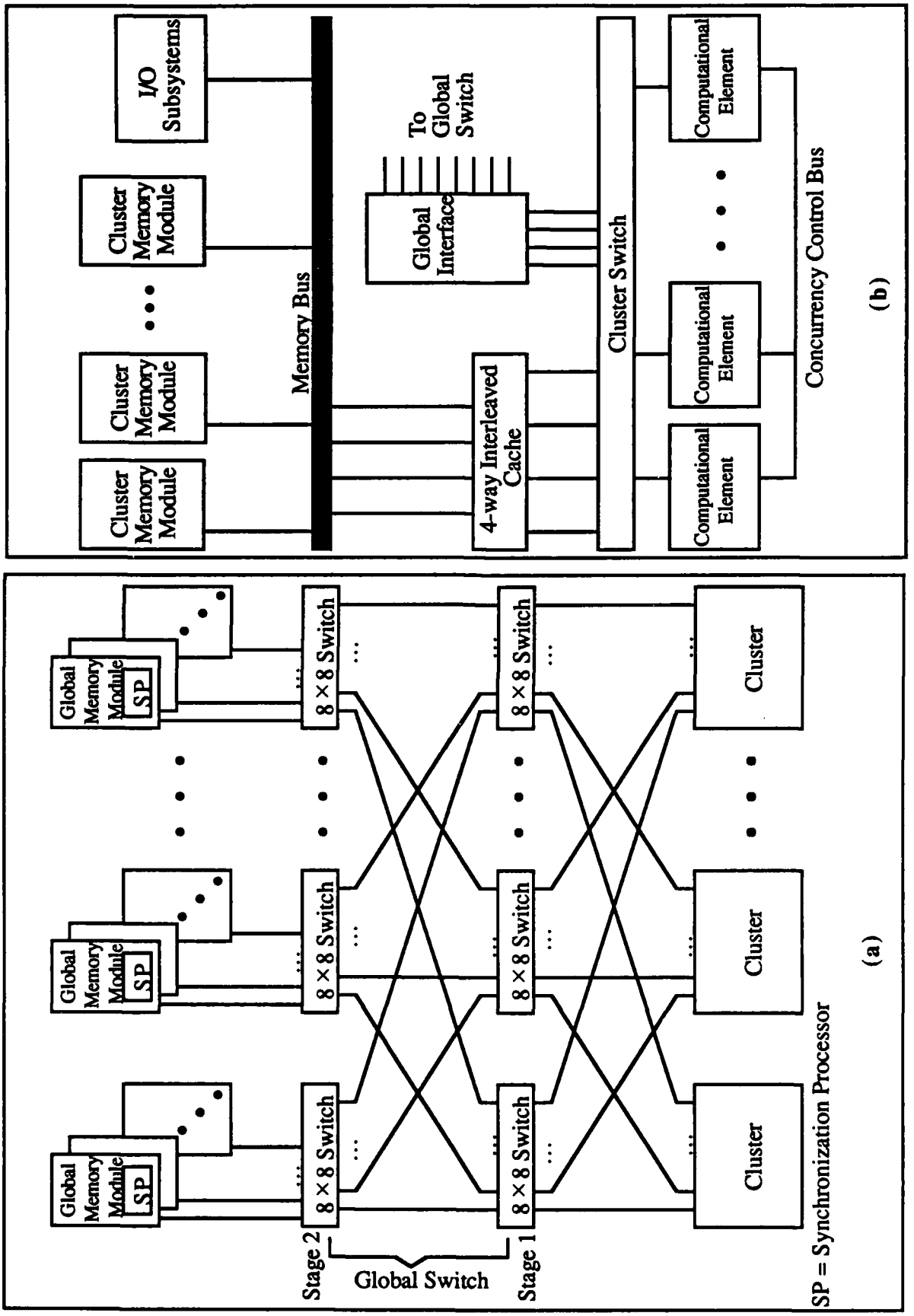


Figure 3-12a. (a) The Cedar System (b) A Cedar Cluster.

C.mmp and Cm*

The following information is based on [Arvind Iannucci 1984], [Charlesworth Gustafson 1986], [Tutorial 1981], and [Tutorial 1984].

C.mmp and Cm* are two multiprocessors developed at Carnegie-Mellon University. C.mmp is a tightly-coupled multiprocessor built on a base of PDP-11 minicomputers connected into a single global memory through a high-speed crossbar switch. Sixteen processors run asynchronously, and can use local memory without interfering with global memory. Interprocessor communication is facilitated by the shared memory and a cross-processor interrupt scheme. Synchronization is performed at several levels due to the recognized need to keep the overhead of this operation very low.

Cm* is the successor to the C.mmp computer. It is a bus-connected system made up of 50 processors. The main difference between the two architectures is that Cm* uses a kind of hierarchical network to interconnect a number of microprocessors, each with its own memory, and Cm* uses packet switching instead of circuit switching in the communication network.

Content Addressable Array Parallel Processor (CAAPP)

The following information is based on [Levitan Weems Hanson Riseman 1987].

The CAAPP is a 512 × 512 square grid array of 1-bit serial processors intended to perform low-level image processing tasks. It is similar to the Loral MPP, but with an architecture that is especially oriented towards associative processing with global summary feedback mechanisms. The processing elements are linked through a four way (north, south, east, and west) communications grid. Each processor contains 192 bits of memory.

The CAAPP is one of the three different, closely coupled parallel processors making up the UMass Image Understanding Architecture at the University of Massachusetts in Amherst, Massachusetts.

Cosmic Cube

The Cosmic Cube is a demonstration 6-dimensional hypercube developed at the California Institute

of Technology (Caltech) by Dr. Geoffrey Seitz. The nodes operate at 0.05 MFLOPS and are based on the Intel 80286 chip. [Charlesworth Gustafson 1986]

DADO

The following information is based on [Stolfo Miranker 1984].

DADO is a large-scale tree-structured parallel machine designed to support the rapid execution of expert systems, as well as multiple, independent systems. The DADO2 prototype, designed at Columbia University, consists of 1023 processing elements (PEs). A full-scale production version of the DADO machine would comprise on the order of a hundred thousand PEs, each containing its own processor, about 16K bytes of local random access memory, and a specialized I/O switch. The PEs are connected to form a complete binary tree (Figure 3-13).

Each PE within the DADO machine is capable of executing in either SIMD mode or MIMD mode, defined as follows. In SIMD mode, the PE executes instructions broadcast by some ancestor PE within the tree. In MIMD mode, each PE executes instructions stored in its local RAM, independently of the other PEs. A single, conventional coprocessor, adjacent to the root of the DADO tree, controls the operation of the entire ensemble of PEs.

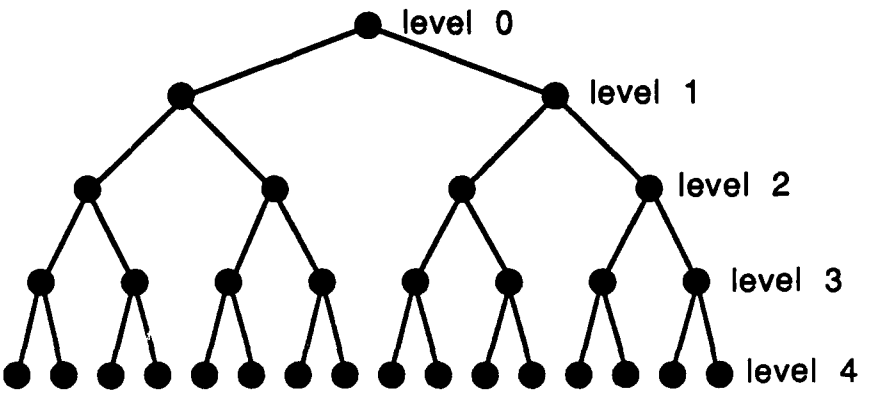


Figure 3-13. A complete binary tree of height 4.

Data-Driven Machine (DDM)

DDM is a dynamic dataflow machine developed by A.L. Davis at the University of Utah. DDM1 is operational and is used for the development of graphical programming languages and to study some

of the basic issues of dataflow. [Tutorial 1984] [Srini 1986]

Dennis Machine

The Dennis Machine is a static dataflow architecture developed at MIT. It is programmed using VAL (Value Algorithmic Language). The Dennis Machine was designed for calculations in weather modeling. [Tutorial 1984] [Srini 1986]

Homogeneous Multiprocessor

The Homogeneous Multiprocessor is a tightly coupled MIMD architecture composed of k ($k \geq 3$) processing elements (PEs), k memory modules, $k + 1$ interbus switches isolating the PEs from each other and the H-network which is a fast local area network used for point-to-point and broadcast mode communications. Each PE has its own local memory module and accesses it via its local bus. The local buses are separated by intervening switches, which provide each PE with the ability to access the memory modules of either one of its two immediate neighbors by requesting the appropriate switch to close. The Homogeneous Multiprocessor system is currently under implementation at the Electrical Engineering Department, Concordia University. [Dimopoulos Li Wong Dantu Atwood 1987]

Manchester Machine

The Manchester Machine is an experimental dynamic dataflow computer constructed at the University of Manchester in England. The dataflow graphs executed by the machine are generated by a compiler from the high-level language Lapse. Lapse is based on a single assignment rule and has the syntax of Pascal. Lapse treats an array as a unit - an assignment to the whole array is done by a single statement. A prototype of the Machine with a single statement pipeline is working at the University of Manchester. [Dennis 1979] [Srini 1986]

NON-VON

The following information is based on Hillyer Shaw Nigam 1986].

NON-VON is a massively parallel non-von Neumann supercomputer, portions of which are being constructed at Columbia University. Figure 3-14 illustrates the top-level organization of the NON-VON machine. In a typical configuration, NON-VON would be connected to a host machine, a general purpose computer serving as a front end device for interactions with the user. NON-VON has two principle components called the *primary processing subsystem* and the *secondary processing subsystem.*

The primary processing subsystem is organized as a binary tree consisting of a large number of small processing elements (SPEs). Each SPE contains an 8 bit ALU, a very small RAM, and communication connections to three neighboring SPEs, the SPE's parent, left child, and right child. Each SPE is also capable of communicating with two additional SPEs (in two clock cycles), the SPE's left neighbor and right neighbor, which are the predecessor and successor of the SPE in an inorder traversal of the primary processing subsystem tree, respectively. SPEs operate synchronously; they receive instructions that are broadcast to them from higher up in the primary processing subsystem tree.

Within the top five to ten levels of the primary processing system, each SPE is connected to a large processing element (LPE). Each LPE is a general purpose microcomputer that can be configured to embody up to a few megabytes of RAM. The LPEs may execute locally stored programs independently and asynchronously. In particular, LPEs at the roots of several subtrees of the primary processing subsystem (possibly at different levels) may broadcast separate instruction streams to the SPEs below them, giving NON-VON the capability for multiple-SIMD execution (see in Section 1-7). The LPEs are connected via a log-stage interconnection network. The exact network has not been decided, but it will be in the butterfly/omega/banyan family and certain configurations based on crossbar switches.

The secondary processing subsystem incorporates between 32 and 256 disk drives, each connected via an *intelligent head unit* to an LPE in the primary processing system. This provides a very high bandwidth interconnection between these two subsystems. In additio to reading and writing data from disks, intelligent head units perform certai.. computationally simple operations, passing results to the associated LPEs.

A prototype primary subsystem containing a single LPE and 8191 SPEs is under construction. This machine will be connected to a VAX 11/750, which will serve as a host.

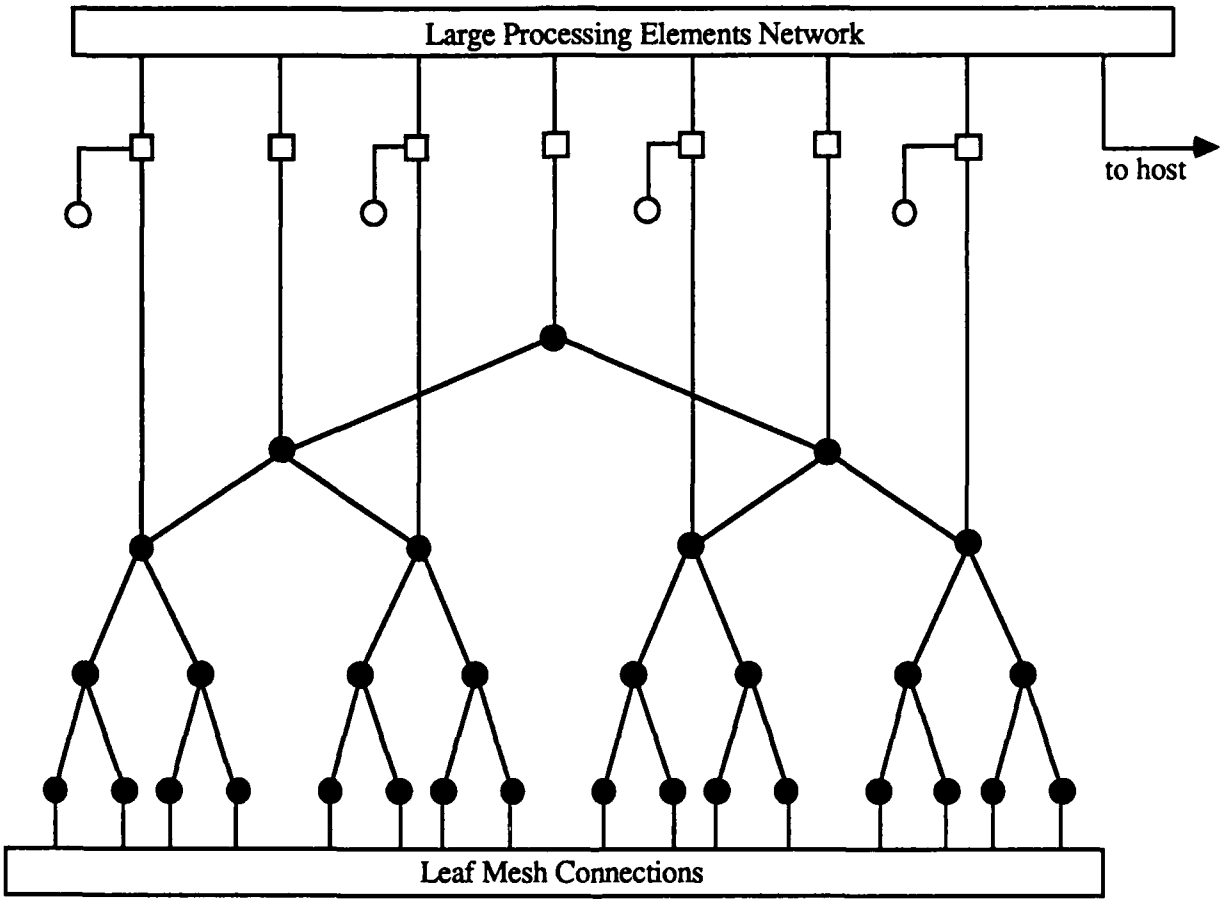


Figure 3-14. Organization of the NON-VON machine.

PASM (PArtionable SIMD/MIMD)

The following information is based on [Casavant Dietz Schwederski Sheu Siegel 1987], [Chu Delp Siegel 1987] and [Siegel Schwederski Kuehn DavisIV 1987].

PASM (partitionable SIMD/MIMD) is a large-scale dynamically reconfigurable multimicroprocessor system. It is a special-purpose system designed to exploit the parallelism of image understanding tasks, but it can also be applied to related areas such as speech understanding and biomedical signal processing. PASM is meant to be a research tool for experimenting with parallel processing.

According to the references given above, a 30 processor prototype of the PASM parallel processing system is near completion at Purdue University. The prototype contains 16 processing elements in the computational engine which may be configured as up to four SIMD clusters of four processors each, a set of independent MIMD processor groups, or any combination of these SIMD/MIMD configurations.

A block diagram overview of PASM is given in Figure 3-15. The *parallel computation unit* (Figure 3-16) contains $N = 2^n$ processors, N memory modules, and an interconnection network. (PASM is being designed for $N = 1024$. A prototype of the PASM system exists for $N = 16$.) The processors are microprocessors that perform the actual SIMD and MIMD computations. The memory modules are used by the processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. A memory module is connected to each processor to form a processor-memory pair called a *processing element* (PE). Each PE can operate in both the SIMD or MIMD modes of parallelism. A pair of memory units is used for each memory module to allow data to be moved between one memory unit and secondary storage (the *memory storage system*) while the processor operates on data in the other memory unit. The *interconnection network* provides a means of communication among the PEs. A generalized cube multistage interconnection network is used in PASM.

The *Micro Controllers (MCs)* (Figure 3-17) are a set of microprocessors that act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. There are $Q = 2^q$ MCs, numbered from 0 to Q-1. Each MC controls N/Q PEs.

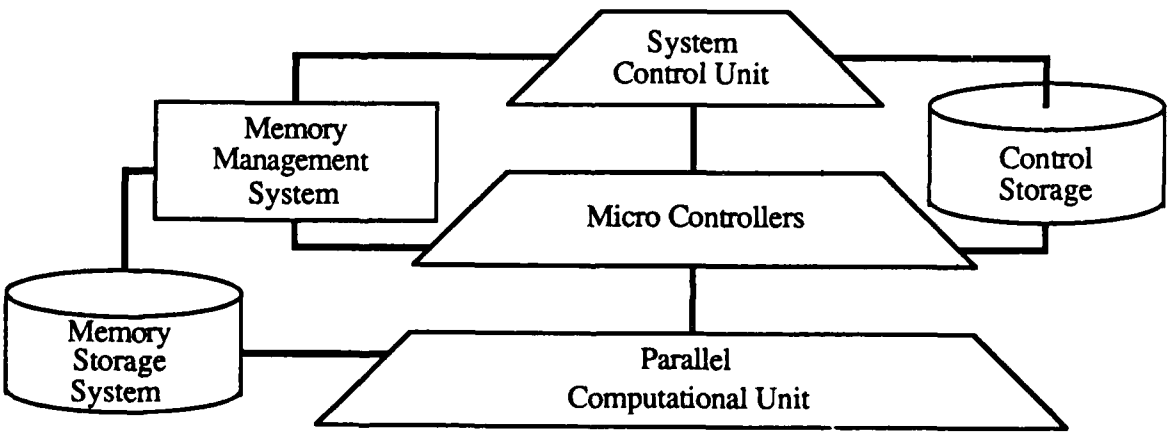


Figure 3-15. A block diagram overview of PASM.

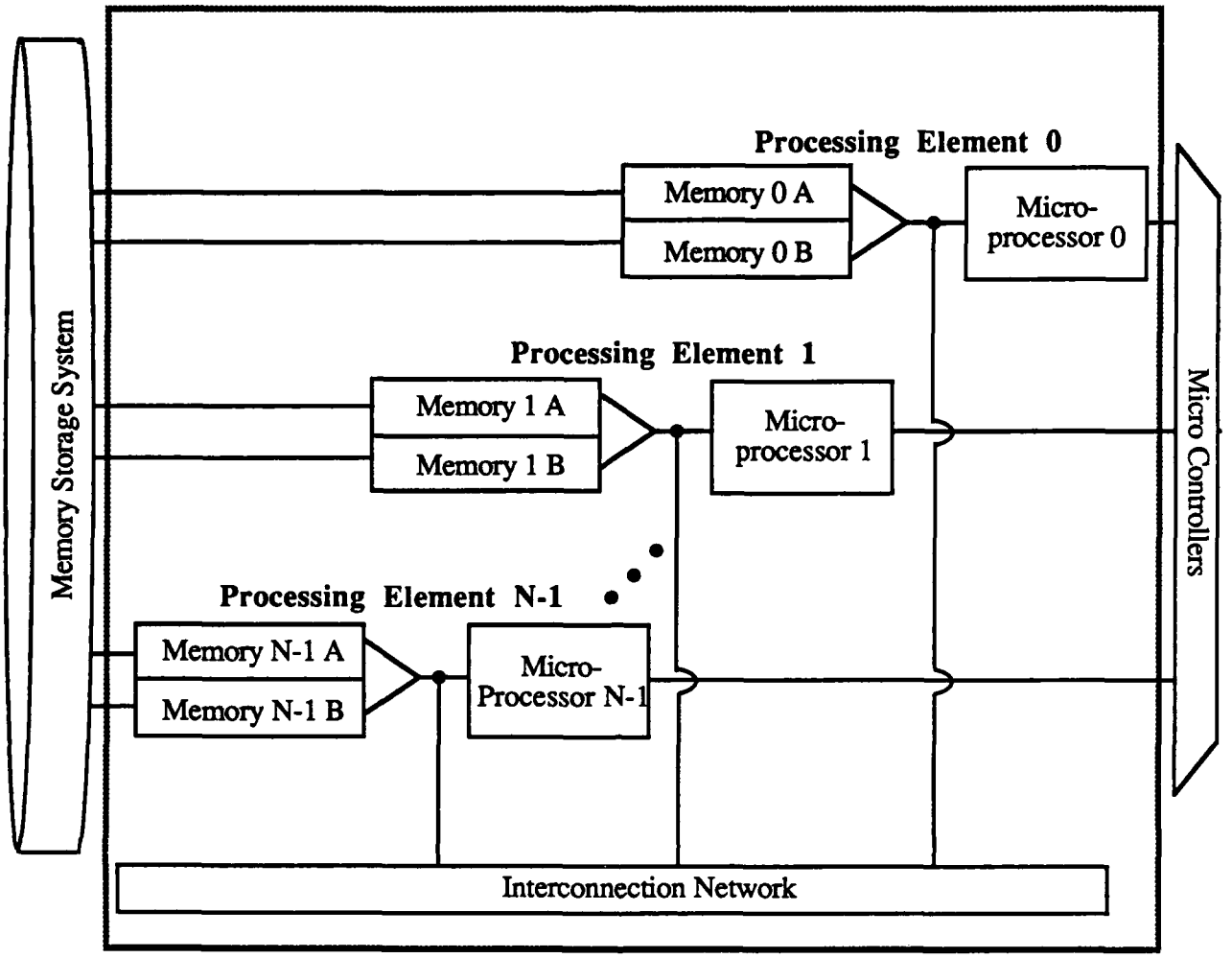


Figure 3-16. PASM Parallel Computation Unit.

The *control storage* contains the programs for the micro controllers. It consists of a secondary storage device and a microprocessor for managing the file system on the device. The control storage acts as a file server by responding to requests to load programs into the micro controller memory unit.

The *memory storage system* (Figure 3-18) provides secondary storage space for the parallel computation unit for the data files in SIMD mode and for their data and program files in MIMD mode.

The *memory management system* controls the transfer of files between the memory storage system and the PEs. It is composed of a separate set of microprocessors dedicated to performing tasks in a distributed fashion. This distributed processing approach is chosen to provide the

memory management system with a large amount of processing power at low cost.

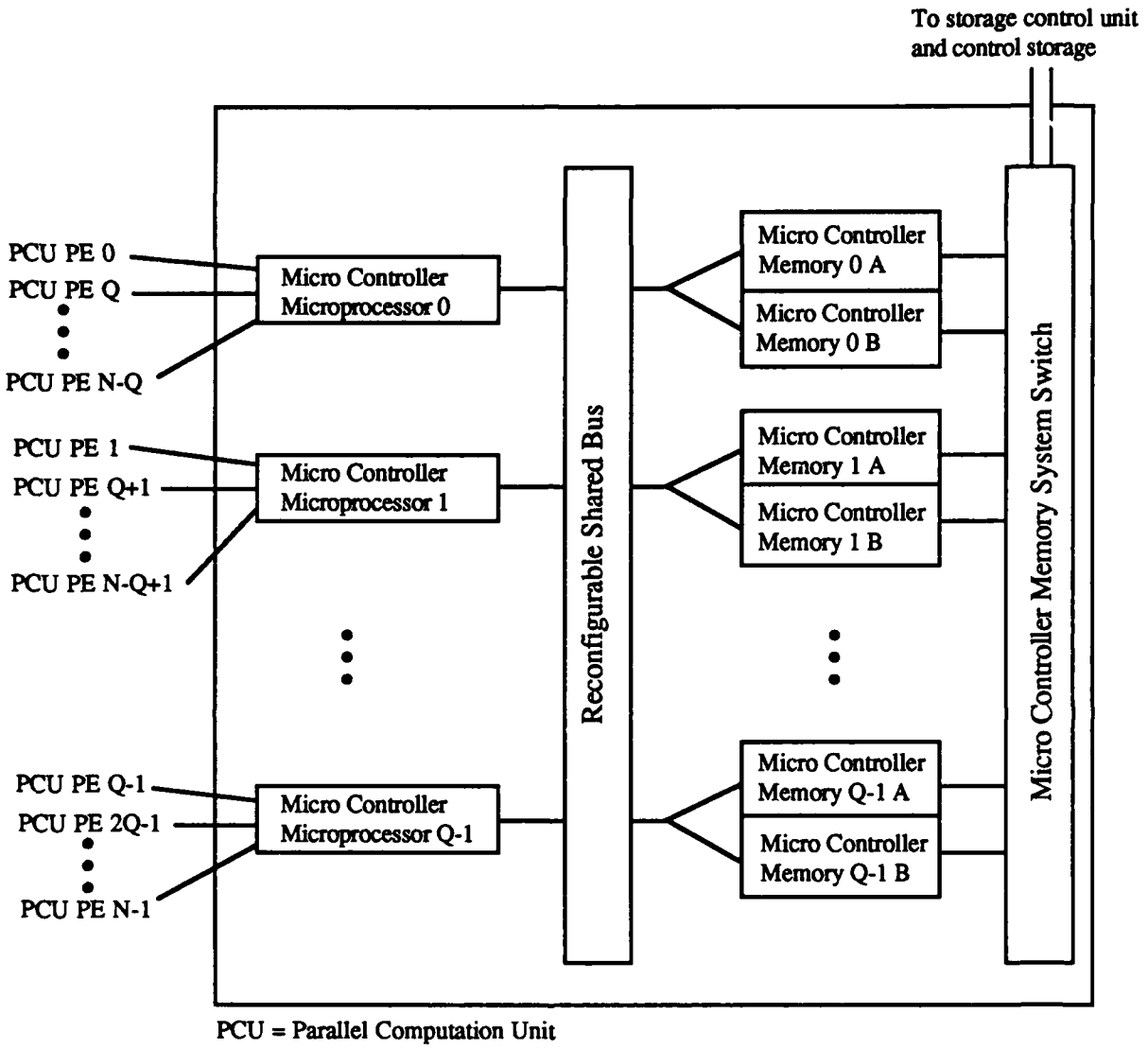


Figure 3-17. PASM Micro Controllers.

The *system control unit* is responsible for the overall coordination of the activities of the other components of PASM. The types of tasks the system control unit performs include program development support, job scheduling (choosing a machine partition for a user job), and coordination of the loading of the PE memory modules from their memory management system with the loading of the micro controller memory modules from control storage.

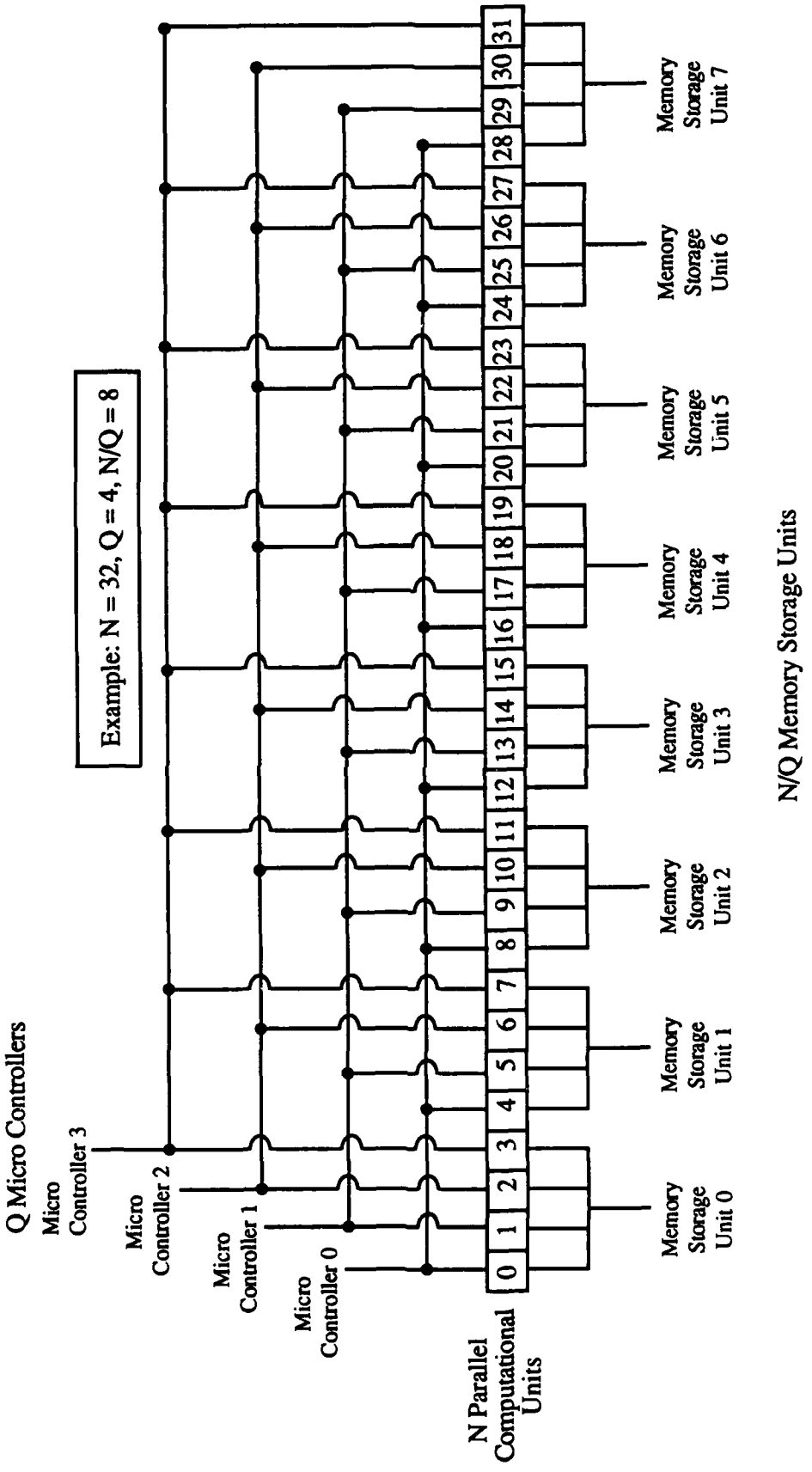


Figure 3-18. Memory Storage System.

The PASM operating system (PASMOS) is distributed among all of the PASM CPUs and is logically divided into two layers: the local kernel layer and the PASMOS (PASM operating system) level. A local kernel is resident on all CPU boards and is responsible for local memory management, local scheduling and synchronization, local I/O device control, local file operations, and reliable communication with other CPUs. PASMOS is a collection of operating system routines distributed among the PASM components. For example, the PASMOS routines for choosing the partition on which a job is to be run exist only in the system control unit. On the other hand, PASMOS routines for handling file load/unload requests from the PEs exist on the micro controllers, the processors of the memory management system, and on the memory storage units.

Shared resources and activities that involve the control of multiple CPUs are the domain of PASMOS. The PASMOS routines use the facilities of the local kernels to perform their functions. User programs make calls to operating system functions in both layers.

A prototype for PASM-2 is under development at the Supercomputing Research Center in Lanham, Maryland.

Research Parallel Processor Prototype (RP3)

The following information is based on [Hwang 1987] and [Pfister Brantley George Harvey Leinfelder McAuliffe Melton Norton Weiss 1986].

The RP3 project is being undertaken by the IBM Watson Research Center in conjunction with the NYU Ultracomputer project. This experimental project aims at investigating the hardware and software aspects of highly parallel computations. The RP3 is an MIMD system consisting of 512 processors with a RISC architecture and a fast interconnection network. Figure 3-19 illustrates the floor plan of the RP3.

The RP3 can be configured as a shared memory system or a message passing system with localized memories or as mixtures of these two paradigms chosen at run-time. In addition, the system can be partitioned into completely independent submachines by controlling the degree of memory interleaving.

The RP3 system will run a modified version of BSD 4.2 UNIX operating system.

Programming languages on RP3 will initially include C, Fortran, and possibly Pascal.

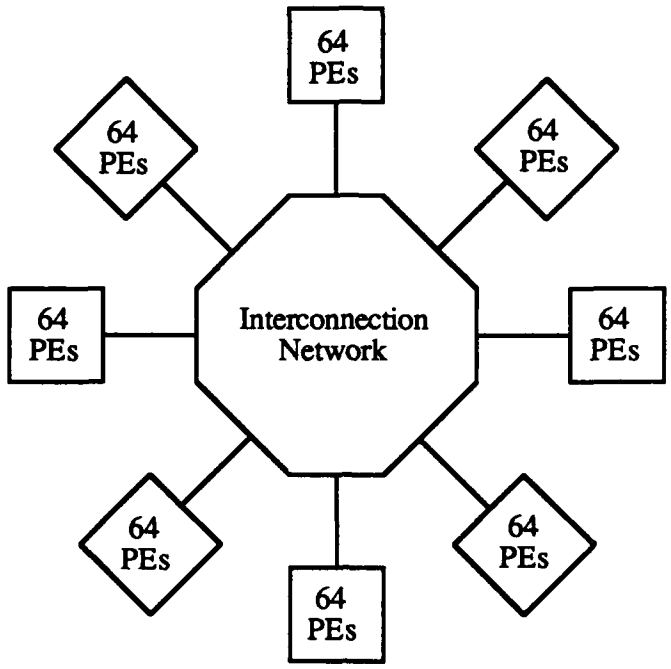


Figure 3-19. The IBM RP3 architecture.

Tagged Token Dataflow architecture (TTD)

The following information is based on [Ghosal Bhuyan 1987] and [Srini 1986].

The tagged token dataflow architecture is a dynamic tagged token dataflow system developed at MIT. The TTD is programmed using the high-level language ID (Irvine Dataflow language). A program graph is generated by a compiler from ID. A host processor loads the program graphs with initial data into the processing elements (PEs) using a scheduler.

The TTD, as shown in Figure 3-20, consists of several physical domains interconnected via an interconnection network. A physical domain consists of a single processing element (PE), a memory controller, and a memory module. The memory controller consists of a complex hardware module which controls the random access memory associated with it. All the memory controllers are interconnected via a single shared bus. The memory in each PE of the dataflow architecture is used to store the dataflow graph as well as for storing structures like arrays and records. The memory in each PE is shared. This allows the memory in each PE to be accessed by other

processors via the shared bus. Further, the shared bus is used to perform all I/O activities.

[Srini 1986] states that an emulation architecture has been in progress using Symbolics Corporation's Lisp machines at MIT. The emulation facility supports 32 PEs and a switch network capable of emulating a variety of interconnection networks. In addition, a variation of the architecture, called SIGMA-1, is under construction. SIGMA-1 supports 256 PEs organized as clusters. Each cluster has an 8×8 crossbar for interconnecting 8 PEs. The clusters are interconnected by a multistage packet switching network using 4×4 switch elements.

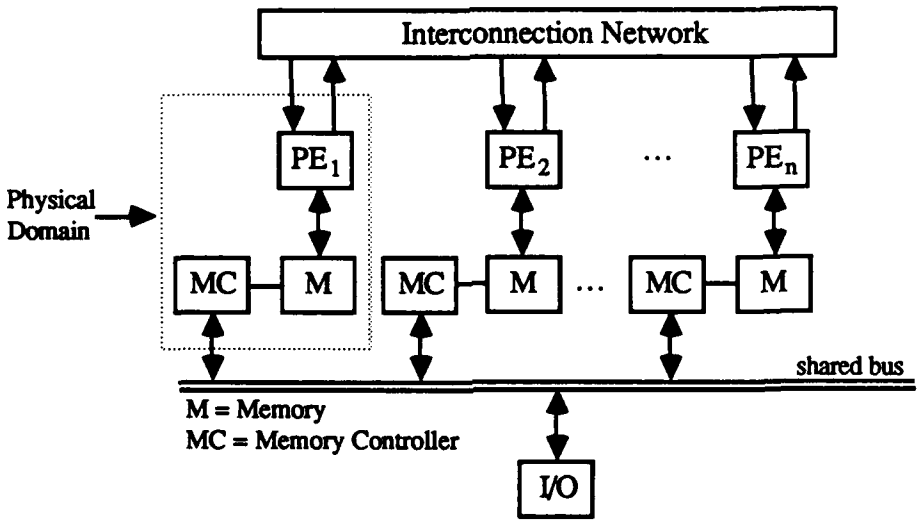


Figure 3-20. MIT Tagged Token Dataflow (TTD) architecture.

Texas Reconfigurable Array Computer (TRAC)

The following information is based on [Deshpande Jenevein Lipovski 1985]

The Texas Reconfigurable Array Computer is an experimental array computer at the University of Texas at Austin. It is expected to be a testbed for parallel algorithms and also a prototype for future general-purpose high-performance computers. TRAC uses a banyan interconnection network, a two-sided, multistage network with processors at the apex end and memories or input/output devices at the base.

Some of the goals of TRAC include the following. TRAC must have an organization to accomodate a large number of processors. It must provide for different modes of communication between the processing units. It must be capable of SISD, SIMD, and MIMD modes of execution.

The system should be dynamically reconfigurable between tasks to support these modes of execution and to maximize the use of system resources. The system organization should be flexible enough to be able to mold the architecture to the algorithm, not the algorithm to the architecture, as has been the case in the past.

Ultracomputer

The Ultracomputer is a shared-memory MIMD computer composed of 4096 autonomous processing elements (PEs). It was developed at New York University. The Ultracomputer employs a message switching network with the geometry of an Omega network. [Gottlieb 1987] [Gottlieb Grishman Kruskal McAuliffe Rudolph Snir 1984]

Warp Computer

The following information on the Warp computer is based on [Annaratone Arnould Cohn Gross Kung Lam Menzilcioglu Sarocky Senko Webb 1987], [Annaratone Arnould Gross Kung Lam Menzilcioglu Webb 1987], [Annaratone Bitz Clune Kung Lam Maulik Ribas Tseng Webb 1987], and [Bruegge Chang Cohn Gross Lam Lieu Noaman Yam 1987].

The Warp computer is a systolic array computer of linearly connected cells, each of which is a programmable processor capable of performing 10 MFLOPS. A typical Warp array includes 10 cells. The Warp machine is integrated as an attached processor to a general-purpose host running the UNIX operating system. Programs for Warp are written in a high-level language called W2, which is supported by an optimizing compiler.

There are three major components in the Warp system, as shown in Figure 3-21: the Warp processor array, the interface unit, and the host. The Warp processor array performs computation-intensive routines such as low-level vision routines or matrix operations. The interface unit handles I/O between the array and the host and can generate addresses and control signals for the Warp processor array. The host supplies data to and receives results from the array. Data flow through the array on two communication channels, X and Y.

The WARP machine is one of the Defense Advanced Research Projects Agency's (DARPA's)

three Strategic Defense Initiative (SDI) machines. The other two are the Connection Machine at Thinking Machines Corporation and the Butterfly at Bolt, Beranek, and Newman, Incorporated (BBN).

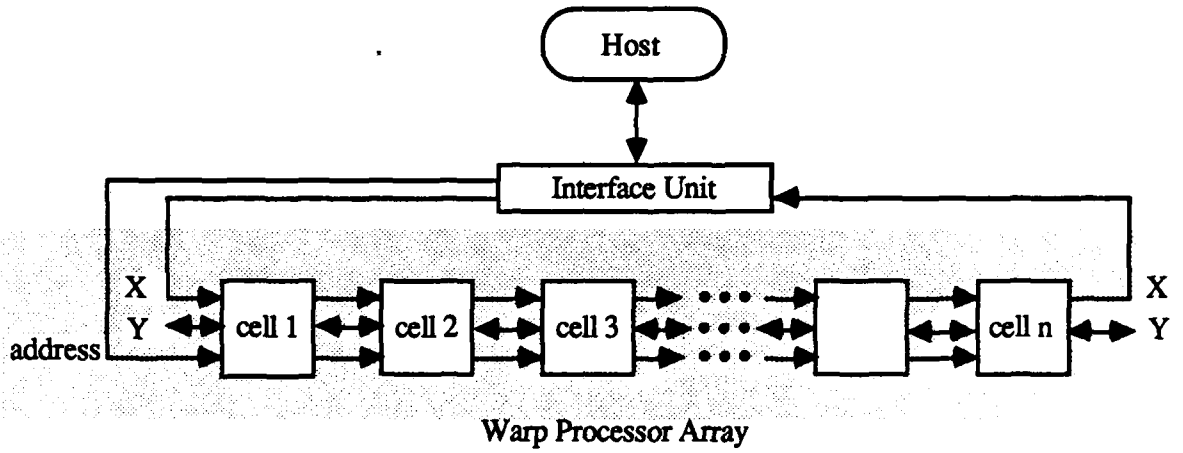


Figure 3-21. Warp System Overview.

CHAPTER 4

LANGUAGES/COMPILERS

Introduction

Due to the fact that Fortran is so widely used in scientific computing, all vendors seem to provide a Fortran-based language for programming their parallel computers. These Fortran extensions are usually based on the underlying hardware and by the capabilities that the vendor feels appropriate for the user, thus leading to varying extensions from one vendor's machine to that of another. Most vendors also provide some extension of the programming language C, and Ada is starting to become available, at least on shared memory machines.

In this chapter, we discuss the various approaches to high-level language development for parallel computers and compare parallel computer languages with conventional serial languages. Software libraries and dataflow computer languages are discussed briefly. Typical vectorization rules which are enforced by automatic vectorizing compilers are discussed. Finally, we give a brief description of a variety of parallel computer languages.

THE CONTENTS OF THIS CHAPTER MAY CHANGE WITH THE NEXT EDITION OF THIS DOCUMENT, DUE TO BE RELEASED IN LATE 1988.

4-1 Language Development Approaches for Parallel Computers

In general, there have been three approaches in developing languages for parallel computers, namely, the *compiler approach*, the *language extension approach*, and the *new language approach*.

The Compiler Approach

Although it is preferable to have vector constructs in high-level languages, most programs written for pipelined vector computers are written in sequential languages. A vectorizing compiler's function is to compile a program written in a sequential language while attempting to extract inherent parallelism and generate vector instructions whenever possible. [Quinn 1987] These compilers include a vectorization phase where Fortran do loops are internally transformed into vector assignment statements. To give the programmer control over what is vectorized and how, these compilers all accept some form of vectorization commands supplied via comment statements. [Padua Guarna Lawrie 1987]

The compiler approach is advantageous for several reasons. First, programmers have the ability to write and debug new code for parallel algorithms in the same style used for sequential algorithms. Second, programmers are able to program supercomputers using familiar languages and are insulated from most of the machine details associated with a choice of efficient parallel implementations. Third, the use of an established sequential language allows the migration of previously written (sequential) applications to parallel computers. [Casavant Dietz Schwederski Sheu Siegel 1987] Last, and probably the main advantage of using standard sequential languages, is portability. Programs, even if they were not written for vector computers, can often be efficiently run on a new vector computer either without change or with the addition of a few compiler directives with vectorization commands for the new machine. [Padua Guarna Lawrie 1987]

There are also several disadvantages to the compiler approach. One disadvantage is that all of the inherent parallelisms of a program are not always detected by the compiler. A programmer cannot code a sequential algorithm in any form and expect to have the vectorizing compiler generate a compiled program that takes full advantage of the power of the computer. (On the other hand, this is true for standard serial programming languages, even with the use of optimizing compilers.) Even though vectorizing compilers are becoming quite sophisticated, the programmer must know something about the underlying architecture. [Quinn 1987] The programmer must assist the compiler by recoding loops to trick it into recognizing them as vector operations. [Allen Kennedy 1982] A second disadvantage is that this kind of language forces the programmer to code parallel

algorithms in sequential form. [Perrott Zarea-Aliabadi 1986]

The Language Extensions Approach

Sequential languages can be extended with architecture-oriented constructs to support parallel programming. These languages are designed to exploit the maximum computing power of the specific hardware architecture to be used. [Ina Kamiya Mikami 1985]

Since the extensions to the languages are machine oriented, they are implemented efficiently. On the other hand, machine dependence implies poor portability. Therefore, when target machines are changed, users may have to recode the parallel algorithm in another extended language. [Hwang 1987] Also, while the syntax of such languages enables the generation of object code best matched to the underlying architecture, it also forces the programmer to have to know the underlying architecture, making program development more difficult. [Ina Kamiya Mikami 1985]

The New Languages Approach

With the new languages approach, new concurrent languages are developed for supporting parallel processing. Occam is an example of such a language. These new languages contain various application-oriented parallel constructs. None of these new languages has been universally accepted in commercial supercomputers. [Hwang 1987]

4-2 A Comparison of Approaches

[SRC 1986b] provides a comparison of the three aforementioned language approaches to conventional (serial) programming languages. This comparison is based on the following criteria: (1) ease of writing code, (2) ease of debugging, (3) ease of modification/maintenance, (4) portability, and (5) attainable efficiency. For this comparison, we recategorize the three language approaches into two categories as follows. The language extensions approach and the new languages approach are both placed in the *explicitly* parallel languages category, and the compiler approach is placed in the *non-explicitly* parallel languages category.

Non-explicitly parallel languages come in three general varieties: (1) languages with unrestricted

side-effects (i.e., conventional sequential languages with whatever parallelism-detection-inhibiting, side-effect-creating constructs they contain in their standard forms), (2) languages with reduced or eliminated side-effects (i.e., functional languages such as Lisp and applicative languages such as BLAZE) in which parallelism detection is relatively simple, and (3) languages with annotated side-effects (i.e., "refined" languages such as refined C and refined Fortran) in which annotation of data-access rights greatly increases the ability to parallelize the code.

Figure 4-1 shows how [SRC 1986b] compares serial programming languages to explicitly and non-explicitly programming languages.

	Explicitly Parallel Languages	Non-Explicitly Parallel Languages
Ease of writing code	Writing code (apart from the question of designing parallel algorithms) is a greater intellectual challenge than writing sequential code.	In languages with unrestricted side-effects or annotated side-effects, coding is as natural as in conventional languages In languages with restricted side-effects, coding is somewhat less natural.
Ease of debugging	Code may be subject to inadvertantly introduced deadlocks and races (updates to shared variables are not sequential in shared memory systems) which render it extremely difficult to debug.	Except in languages with restricted or eliminated side-effects, debugging is purely conventional. Debugging methods and tools can be carried directly over.
Ease of modification/ maintenance	Modification and maintenance by other than the original programmer is prone to deadlocks and races.	Code modification and maintenance involve the same effort as for sequential languages.
Portability	If a new target architecture is very different from the original target, porting can require rewriting.	Portability is not the concern of the programmer. It is the concern of the compiler-writer who writes a new back-end for each drastically different architecture.
Attainable efficiency	Since architecture-dependence is typically far greater for parallel machines than for sequential machines, greater intellectual effort is required to attain efficient execution.	Optimized speed-up is usually not readily obtainable, but quite reasonable speed-up is available to a much wider user community.

Figure 4-1. A comparison of sequential languages to explicitly and non-explicitly parallel languages.

4-3 Libraries

A popular way to aid parallel computer programmers is to provide them with libraries. That is, a

variety of computational blocks of algorithms highly suited for the hardware architecture are designed and then provided to the programmer as a library. For example, the programmer can use the library by means of Fortran call statements. Examples include Scientific Subroutine Library (SSL) II/VP from Fujitsu, EISPACK from Intel Scientific Computers, CRAYPACK from BCS Company, APMATH from Floating Point Systems, Ltd. and MATRIX/HAP from Hitachi. Some libraries only cover fundamental functions such as matrix or vector operations, while others cover an extensive scope of applications including fast Fourier transforms, sorting, linear equations, algebraic eigenvalue problems and differential equations. The only apparent problem with these libraries is that the programmer may have to adjust his/her program to the library interfaces. [Ina Kamiya Mikami 1985]

4-4 Automatic Vectorizing Compilers

The following information is based on [Bossavit Meyer 1981], [Padua Guarna Lawrie 1987], [Padua Wolfe 1986], and [Soll 1986].

In order to be vectorizable by an automatic vectorizing compiler, a program must satisy certain conditions. Five basic conditions appear as necessary and sufficient in most vector computers (not including machine peculiarities):

(1) *Repetitive series of operations.* Counter loops with the number of executions known at the outset (e.g., for loops in Pascal and do loops in Fortran) are the **only** sequences amenable to vectorization. Since not all loops are do loops, the programmer must recode those which are not do loops in the form of a do loop in order for it to be vectorizable by the compiler.

(2) *Primitive operations only.* A vector loop can only contain assignments and numerical or boolean operations (some slight extensions do exist). Inhibitors vary from machine to machine, but usually include subroutine calls, I/O statements, conditional statements (if/then), goto statements and other loops.

(3) *Regularity*. In order to be vectorizable, a for loop must involve only array elements whose indices follow a strictly defined pattern, thus allowing them to be fetched in advance for vector operations.

(4) *No backward dependence*. Let a loop with i as a counter contain the following array element assignment:

$$a[f_0(i)] \leftarrow op(b_1[f_1(i)], b_2[f_2(i)], \ldots, b_m[f_m(i)])$$

where square brackets are used for array elements, ***op*** is some numerical or logical operation, the f_k's are linear functions, and all arrays are considered as one-dimensional.

This assignment has a backward dependency if and only for some k $(1 \leq k \leq m)$ b_k is a, and for some pair of values p, q, in the range of i, the following holds:

$$p < q \textbf{ and } f_k(p) = f_0(q).$$

In other words, the computation of $a[f_0(q)]$ will use the value of another element of a, which was fetched for updating in some previous iteration. For example, the assignment

$$a[i] \leftarrow a[i - 1] + 1$$

introduces a backward dependency.

This rule is necessary because the vector interpretation of such a computation would use the old value of the array element, not the new one as in the standard (sequential) interpretation of the loop.

(5) *No cross dependency*. Suppose a loop contains the following assignments:

$$a[f_0(i)] \leftarrow \mathbf{op}(\ldots);$$
$$c[g_0(i)] \leftarrow \mathbf{op'}(\ldots, a[g_j(i)], \ldots)$$

They induce a cross dependency if and only if for some pair of values p, q in the range of i, the following holds:

$$g_j(p) = f_0(q)$$

with $|q - p| < 64$ (where 64 is chosen as an example; 64 is the length of the vector registers on the Cray-1).

For example, the following statements in a loop on i will cause a cross dependency:

$$a[i] \leftarrow 1;$$
$$c[i] \leftarrow a[i + 1].$$

This rule is necessary due to the limited size of the instruction buffers. Long loops may have to be split into several shorter ones in order to be vectorized (e.g., by slices of 64 on the Cray). This would mean that the two assignments might end up in two different loops, thus changing the semantics of the program.

4-5 Dataflow Computer Languages

The following information is based on [Ackerman 1979] and [Haynes Lau Siewiorek Mizell 1982].

Dataflow machines demand high-level languages since graphs, their machine language, are not an appropriate programming medium. Graphs are error-prone and hard to manipulate. Three high-level language classes have been considered by dataflow researchers as follows:

(1) *Imperative class*. This class includes languages such as Pascal, Fortran, and Ada, which change the value of variables via assignment statements. For example, the Texas Instruments group considered the use of a modified ASC Fortran compiler for their dataflow machine

[Jensen 1980]. Compiler techniques for the translation of imperative high-level languages into dataflow graphic languages have been studied at Iowa State University [Allan Oldehoeft 1979].

(2) *Functional class.* This class includes languages resembling Lisp. Scientists at the University of Utah have done research on this class [Keller Jayaraman Rose Lindstrom 1980].

(3) *Dataflow languages.* This class of languages is designed with dataflow machines in mind. Examples include *Id* [Arvind Gostelow Plouffe 1978] [Arvind Thomas 1980], *LAU* [Conte Hifdi Syre 1980], and *VAL* [Ackerman Dennis 1979]. The syntax of dataflow languages is similar to that of imperative languages. For example, all dataflow languages include if and loop statements. On the other hand, their semantics are basically that of functional languages.

As discussed above (and in Section 1-13), the machine level program of a dataflow computer is represented in the form of a graph with pointers between nodes, the pointers representing both the flow of data and the sequencing constraints. Each instruction is kept in a hardware device (an extremely simple processor) that is capable of "firing" or executing an instruction when all of the necessary data values have arrived, and sending the result to the processors that hold destination instructions. Hence, the programming language for a dataflow computer must satisfy the following criteria:

(1) It must be possible to deduce the data dependencies of the program operations.

(2) The sequencing constraints must always be exactly the same as the data dependencies, so that the instruction firing rule can be based simply on the availability of data.

The corresponding properties of a language which make it possible to meet these criteria are locality of effect (i.e., instructions do not have unnecessary far-reaching data dependencies) and freedom from side effects.

4-6 A Sampling of Parallel Computer Languages

This section discusses a variety of programming languages which have been designed for programming parallel computers. Programming languages which have automatic vectorizing compiler counterparts are not discussed because they are similar to each other in that they all enforce some set of rules of vectorization on the programmer. That is, vector computers are typically programmed using some Fortran-based language with an associated automatic vectorizing compiler which takes advantage of the particular underlying architecture. Section 4-4 already discussed the typical rules of vectorization enforced by these automatic vectorizing compilers. It would be redundant to list the vectorization rules of all the vectorizing compilers which have been developed for vector computers.

Actus II

The following information is based on [Perrott Lyttle Dhillon 1987].

Actus II is a Pascal-based parallel language designed for programming processor arrays. Actus II is a refinement of Actus [Perrott 1979] [Perrott Crookes Milligan 1983]. It was designed specifically for those processor arrays which offer a grid of processing elements to perform the computation.

Actus II is currently being implemented on the ICL Distributed Array Processor (DAP), an SIMD mesh-connected computer with 4096 processing elements (64×64). Until the implementation of Actus II, the only language available on the ICL-DAP was DAP-Fortran, the syntax of which reflects the architecture. Unlike DAP-Fortran, Actus II allows the expression of a parallel algorithm independent of the number of processing elements, thus enhancing program portability.

The extensions made to standard Pascal include data definitions and language constructs to facilitate parallel processing. Figure 4-2 illustrates some of the features of Actus II. The Actus II compiler is constructed to be as independent as possible of the underlying architecture while at the same time producing efficient object code. It has been organized so as to facilitate implementation on different processor arrays.

BLAZE

The following information is based on [Koelbel Mehrotra van Rosendale 1987] and [Mehrotra van Rosendale 1987].

BLAZE is a block-structured scientific parallel programming language with a Pascal-like syntax designed to simplify the task of programming multiprocessor parallel architectures. It contains extensive array manipulation facilities. Control flow in BLAZE is entirely sequential with the exception of the forall statement. A central goal in the design of BLAZE is portability across a broad range of parallel architectures (both SIMD and MIMD).

A Sampling of the Features of Actus II	For example:
Parallel Arrays: To introduce parallelism into into an array declaration, one or two pairs of parallel dots (:) are used in place of the usual sequential dots (..).	**var** ParallelArray: **array** [1:n] **of** integer; This declaration defines a one-dimensional parallel array of *n* elements, all the elements of which can be accessed simultaneously.
Index Sets allow simultaneous access to all or selective elements of a parallel variable. In the examples to the right, "Range" defines a constant index set representing the parallel range 1:20, and "IndexSet" defines an index set which can take on any values allowed by the type integer.	**index** Range: 1:20; IndexSet: integer; given the declaration **var** A: **array** [1:50] **of** integer; A[Range] will access the first 20 elements of the array simultaneously.
Parallel Statements: In order to facilitate the construction of parallel algorithms for processor arrays, program structures assignment, **if**, **case**, **while**, procedure and function abstractions allow the manipulation of two dimensions of parallelism.	**var** A: **array** [1:25, 1:100] **of** integer; **index** RowIndex: 1:25; ColumnIndex: 1:100; The statement A[RowIndex, ColumnIndex] := 1; assigns the value 1 to all 2500 elements of array A simultaneously

Figure 4-2. Sample features of Actus II.

Concurrent Pascal

The following information is based on [Brinch Hansen 1975].

(Although Concurrent Pascal is not a language designed for programming parallel computers, it has been included here to distinguish it from Parallel Pascal, which people often confuse as being the same as Concurrent Pascal.)

Concurrent Pascal was developed specifically for the structured programming of computer operating systems. Concurrent Pascal extends the sequential programming language Pascal with concurrent programming tools called *processes* and *monitors*.

A *process* consists of three items: (1) a *private data structure*, (2) a *sequential program* that can operate on the data, and (3) the process' *access rights*, which indicate the shared data it can operate on. A process cannot operate on the private data of another process, but concurrent processes can share certain data structures (e.g., a disk buffer).

A *disk buffer* is a data structure shared by two concurrent processes. A process should only use a disk buffer to send and receive data. Misuse of a shared data structure through either a programming mistake or tricky programming should be detected by a compiler. A *monitor* is a language construct which allows a programmer to tell a compiler how a shared data structure is to be used by all processes. With this knowledge, a monitor is able to synchronize concurrent processes, transmit data between them, and control the order in which competing processes use shared, physical resources. A monitor consists of four items: (1) a *shared data structure*, (2) *monitor procedures* (all operations that processes can perform on it), (3) an *initial operation* that will be executed when its data structure is created, and (4) *access rights*.

Extended Ada

[Cline Siegel 1985] presents a minimal set of features which make Ada suitable for use with SIMD type architectures. Machine independent constructs are proposed the make the language specified very general. The intent was not to propose a language for a particular architecture, but rather one which is of use in the specification of problems for many different architectures. The language is intended to be applicable to a variety of both SIMD architectures and SIMD algorithms. The

intention of Ada to be portable, readable, and widely applicable is also the intent of the parallel language discussed.

Fortran 8x

The following information is based on [Metcalf 1987].

The Fortran standardization committee, X3J3, has been working on the next Fortran standard, Fortran 8x, since 1979. Fortran 8x was originally supposed to be completed by 1982, but current projected development completion is 1989. The new standard will then finally be changed from its working name Fortran 8x to Fortran 88 (cheating just a little).

Fortran 8x will incorporate powerful array processing features and derived-data types, allowing users to access vector processor hardware using a conventional notation, and to define and manipulate objects of their own design. Of course, some features will be also be removed from the language. Otherwise, the language will get too large and contain many overlapping and redundant items.

A few new features included in Fortran 8x are as follows. The new source form allows free form source input, without regard to columns. Comments may be in-line, following an exclamation mark (!), and lines which are to be continued bear a trailing ampersand (&). The character set is extended to include the full ASCII set, including lower-case letters. The underscore character is accepted as part of a name, which may contain up to 31 characters. An alternative set of relational operators is introduced, namely, < for **.LT.**, > for **.GT.**, <= for **.LE.**, >= for **.GE.**, == for **.EQ.**, and <> for **.NE.**. The `case` construct has been added, which allows the execution of one block of code, selected from several, depending on the value of an integer, logical, or character expression. The most important new aspects of Fortran 8x are the array processing features. The operations, assignments and intrinsic functions are extended to apply to whole arrays on an element-by-element basis, provided that when more than one array is involved they are the same shape. Fortran 8x also provides four separate mechanisms for accessing storage dynamically. About 100 intrinsic procedures are defined, many intended for use in conjunction with arrays for the purposes of reduction (e.g., `sum`), inquiry (e.g., `rank`), construction (e.g., `spread`),

manipulation (e.g., transpose), and location (e.g., maxloc). User-defined data types will be allowed in Fortran 8x.

Glypnir

The following information is based on [Welch 1984].

Glypnir is an Algol-based block structured language designed for programming the Illiac IV. The mesh interconnection network between the PEs in the Illiac IV is embodied in Glypnir to exploit the machine capability. Glypnir is block structured in that it allows a single statement to be replaced with a block of statements delimited by begin and end.

In Glypnir, vectors are arrays of memory elements that must be declared. For example, the declaration PE REAL VECTOR A[10] declares *A* to be a real vector (for a processing element's use) with 10 rows of 64 words per row. The declaration CU REAL VECTOR B[10] declares *B* to be a real vector (for the Illiac IV control unit's use) with 10 rows of 1 word per row.

Assignment statements transfer values from one memory location to another. For example, in the general assignment statement X := [R]Y + Z, *R* is the routing index and may be a general arithmetic expression whose result is a positive or negative integer, where positive values specify a right- and negative a left-routing procedure. The quantity $Y + Z$ is computed in the enabled PE *R* to the right (or left if negative) of the PE currently being addressed. The result $Y + Z$ is then routed back to the addresses PE.

Multi-Pascal

Multi-Pascal is an extension of Pascal with the addition of new features for creating concurrent processes and for process communication, using either shared memory or communication channels. Multi-Pascal has three types of primitives for creating concurrent processes: the $ operator, cobegin/coend and forall. The $ operator may appear at the beginning of any statement and causes that whole statement to become a detached parallel process running concurrently with its creator. Cobegin/coend is used to surround a list of statements causing each individual statement in the list to become a concurrent process. Unlike the $ operator, cobegin/coend makes the

originating program wait until all the processes terminate before it can continue. The forall instruction is a parallel form of a normal for loop, and is used for highly parallel vector or array operations. [Lester Guthrie 1987]

Occam

The following information is based on [Hull 1987], [Wayman 1986] and [Whitby-Strevens 1985].

Occam is a programming language designed by Inmos Limited to support software development on the Inmos transputer. The transputer is a microprocessor which has been uniquely designed to function as a component processor in a network or array of multiple processors. Together, the Inmos transputer and the occam programming language provide a method for designing and implementing systems made up of communicating processes. Occam's notation is based on Hoare's Communicating Sequential Processes (CSP) [Hoare 1978].

The basic unit of an occam program is the *process*. Occam enables a multi-transputer system to be described as a collection of processes that operate concurrently and communicate using message passing via named channels. Three primitive processes are used to build occam programs:

1. *variable* := *expression*	assign *expression* to *variable*
2. *channel* ! *expression*	output *expression* to *channel*
3. *channel* ? *variable*	input from *channel* to *variable*

These primitive processes are combined to form *constructs*. Each construct is introduced by a keyword, followed by a list of the component processes:

SEQuential	components executed one after the other
PARallel	components executed together
ALTernative	component first ready is executed

A construct is itself a process, and may be used as a component of another construct. Conventional sequential programs can be expressed with variables and assignments, combined in sequential constructs. if and while constructs are also provided.

Concurrent programs can be expressed with channels, inputs and outputs, which are combined in parallel and alternative constructs. Each occam channel provides a communication path between two concurrent constructs. Communication is synchronized and takes place when both the inputting process and the outputting process are ready. The data to be output is then copied from the outputting process to the inputting process, and both processes continue.

Parallel Pascal

The following information is based on [Reeves 1984].

Parallel Pascal extends the sequential programming language Pascal with a convenient syntax for specifying array operations. Parallel Pascal was originally designed as a high-level programming language for NASA's Massively Parallel Processor (MPP), which was constructed by Loral Systems (formerly Goodyear Aerospace). The MPP is an SIMD mesh-connected computer with 16,384 processing elements (the MPP is discussed in more detail in Section 3-1). Parallel Pascal is particularly suitable for the SIMD class of computers, but Anthony Reeves (designer of Parallel Pascal) recently started looking into implementing Parallel Pascal on the FPS T-Series hypercube (an MIMD machine) [Reeves Bergmark 1987].

Parallel Pascal provides three fundamental classes of operations on array data as primitives of the language: (1) *data reduction* operations, (2) *data permutation* operations, and (3) *data broadcast* operations. Parallel Pascal defines a parallel control statement called the where statement. This statement is similar to an if statement, but with an array control variable. Parallel Pascal also provides a method of accessing the individual bits of array data elements. Figure 4-3 illustrates some of these features.

ParMod

The following information is based on [Eichholz 1987].

ParMod is an extension of the programming language Pascal designed to take advantage of the concept of modules, a concept which has proven to be essential in the field of software engineering. The main principle of ParMod is PARallel execution of MODules. ParMod is designed

to make use of concurrency. Therefore, the hardware system which is to run a ParMod program should contain several autonomous, communicating computational units.

A ParMod program consists of several modules which are executed in parallel. Communication between modules is only performed by calling global procedures. No variables may be common to different modules, but tasks within modules may use common variables which are declared locally in that module. ParMod provides parallelism within a module as well as parallelism between modules.

PFP (Parallel Fortran Prototype)

IBM developed PFP as part of a joint study between IBM and the Cornell National Supercomputer Facility (CNSF). PFP is an extension to VS Fortran which allows for parallel execution at both the do loop level and the subroutine or task level. In addition to the user-specified constructs, there is a facility for automatic parallelism. There is no need to choose between use of parallel or vector execution with PFP - the user can take advantage of both in order to decrease the turnaround time of engineering/scientific computations. [Forefronts 1988]

Refined Languages

[Dietz Klappholz 1985] present a methodology which permits any conventional, sequential language (e.g., C, Fortran, Ada, Pascal or PL/I) to be modified so that

(1) Users can write high-level language code which differs from conventional code only in that data access rights are more precisely specified

(2) Compilers, using well-known flow-analysis techniques, can generate consistently good, highly parallel, race-free code for virtually any machine architecture.

The resulting language is called a refined language (e.g., Refined C or Refined Fortran). The goal of this methodology is to provide a more general way of expressing algorithms for parallel computers without imposing a different programming style.

The transition from a conventional sequential language to a refined language is as follows. Remove from the conventional language any constructs which support the explicit specification of parallel execution and interprocess communication and synchronization. This is done in order to remove the possibility of writing a race condition. Next replace any remaining constructs which deter a flow-analyzing compiler from being able to produce highly-parallel code with modified versions which do not inhibit analysis, yet provide nearly all of the expressive power of the constructs they are replacing.

Vector C

Vector C is a superset of conventional C which has been designed and implemented on the Cyber 205. The syntax of Vector C allows for the easy, natural expression of vector algorithms in a direct manner. The extensions made to conventional C include vector data types, vector expressions, vector operators and/or keywords and built-in functions. Figure 4-4 illustrates some of these extensions. [Li Schwetman 1985]

VECTRAN

VECTRAN was developed within IBM in the early 1970's as an experimental language extension to Fortran to study and facilitate the introduction of vector/array and parallel processing algorithms in scientific and engineering application programs. Since its publication in 1975, VECTRAN has been used as a functional model for development of several vector/array extensions to Fortran including the new proposals by the American National Standards Institute X3J3 Committee for the future ANSI Fortran standard, Fortran 8x. [Paul 1984]

A Sampling of the Features of Parallel Pascal	For Example

Parallel expressions: All conventional expressions are extended to array data types.	**var** a, b, c: **array** [1..20] **of** integer;
	
	The statement
	a := b + c + 5;
	is valid and is equivalent to
	FOR i ←1 TO 20 DO
	a[i] ← b[i] + c[i] + 5;

Reduction functions: Array reduction operations are achieved with a set of standard functions: sum, prod, all, any, max, and min. The first argument of each function is the array to be reduced and the following *n* arguments specify which dimensions are to be reduced. (Functions "all" and "any" reduce arrays with boolean operators AND and OR, respectively.)	**var** a: **array** [1..10,1..5] **of** integer; b: **array** [1..10] **of** integer; c: integer; b := sum(a, 2); sum the rows of a c := max(b, 1); find the max value of b

Permutation and Distribution Functions:
shift(array, S1, S2, ..., Sn):
end-off shift data within "array" by the amount specified for each dimension and shift zeros in at the edges of "array"
 	
rotate(array, S1, S2, ..., Sn):
circularly rotate data within "array" by the amount specified for each dimension
 	
transpose(array, D1, D2):
transpose two dimensions of "array"
 	
expand(array, dim, range):
increase the rank of "array" by one by repeating the contents of "array" along a new dimension as specified by the second parameter
	var
	a,b: **array** [1..5, 0..9] **of** integer;
	c,d: **array** [0..9] **of** integer;
	
	the statement
	c := rotate(d, 3);
	is equivalent to
	FOR i ←0 TO 9 DO
	c[i]←d[(i + 3) mod 10];
	
	the statement
	a := b + expand(c, 1, 1..5);
	is equivalent to
	FOR i ←1 TO 5 DO
	a[i,]←b[i,] + c

Conditional Execution:
 	
where array-expression **do**
statement
otherwise
statement
 	
This control structure allows the programmer to operate on a subset of the elements of an array.
	var
	a, b, c: **array** [1..10] **of** integer;
	
	the expression
	where a < b
	c := b
	otherwise
	c := a;
	is functionally equivalent to
	FOR i ←1 TO 10 DO
	IF a[i] < b[i] THEN
	c[i]←b[i]
	ELSE
	c[i]←a[i]

Figure 4-3. Sample features of Parallel Pascal.

A Sampling of the Features of Vector C:	For example:
Implicit Vector Data Types: Vectors are declared as in the conventional C array declarations. In the examples, *va* is a floating vector of length 100, *vb* is an integer matrix of size 10× 20, and *vc* is a character vector of length 50.	**float** va[100]; **int** vb[10][20]; **char** vc[50];
Explicit Vector Data Types: A vector is explicitly declared using the **vector** (a new keyword) storage class. In the examples, *v* and *m* are vector arrays (i.e., vectors which are supposed to be processed by the vector processor), and *a* and *b* are sequential arrays (i.e., they are supposed to be processed by the scalar processor).	**vector float** v[100], m[10][10]; **float** a[100], b[10][10];
Vector References: The examples illustrate only a few of the ways to reference vectors.	m[*][i] the ith column of matrix m m[*][*] the entire matrix m v[i:f] a (sub)vector consisting of elements v[i] through v[f] v[0#4] initial # length (i.e., v[0], v[1], v[2], v[3])
Vector Expressions: A vector expression is an expression which contains at least one vector reference. For the example, the declarations are as follows: int va[100], vb[100];	va[0#n] = vb[3#n] + c which is equivalent to FOR i ←0 TO n - 1 DO va[i] ← vb[3 + i] + c

Vector Operators: "@+" is the reduction operator for vector sum. "@<" is the reduction operator for vector minimum.	**Vector sum:**
	sum = @+ va[0#100];
	which is equivalent to
	sum ← 1;
	FOR i ←1 TO 99 DO
	sum ← sum + va[i]
	Vector Minimum:
	min = @< va[1:100];
	which is equivalent to
	min← va[1];
	FOR i ← 2 TO n - 1 DO
	if va[i] < min then
	min ← va[i]

Figure 4-4. Sample features of Vector C.

CHAPTER 5

COMMUNICATION/SYNCHRONIZATION

Introduction

The information in this chapter is based on [Howe Moxon 1987], [Hwang Briggs 1984], [Karp 1987], [Quinn 1987] and [Sonnenschein 1986].

In order for processes to work together, they must have the ability to communicate and synchronize. Processes communicate either through shared variables (in shared memory systems) or through message passing (in distributed memory systems).

Communication typically leads to requirements for synchronization. There are two uses of synchronization: (1) to constrain the ordering of events and (2) to control interference. An example of the first kind of synchronization is a mechanism which prevents a process in a pipelined parallel algorithm from writing into a full buffer or from reading from an empty buffer. An example of the second kind of synchronization is the use of a lock statement to prevent a process from accessing the value of a variable while another process is updating it. The lock statement is discussed in more detail in Section 5-2.

5-1 Communication and Synchronization in Distributed Memory Systems

Message passing is a form of communication, since a process receiving a message is receiving data from another process. Message passing is also a form of synchronization, since a message can be received only after it has been sent. (Note that due to the current state of algorithm development, it

is typically the case that for distributed memory, message passing systems, there is at most one process per processor.)

Two functions most important to distributed memory, message passing systems are *send* and *receive*. Send is used to send a message from one process to another. Two forms of send are (1) an *unblocked* or *asynchronous send*, which continues processing immediately after dispatching the message, and (2) a *blocked* or *synchronous send*, which waits to make sure the message has arrived (but not necessarily been read into the local memory of the recipient) before continuing. Typical arguments in a send statement include the message length, the message itself, the label of the destination processor, and the identity of the destination processor (if more than one process exists per processor). Other arguments often used include the message type being sent, routing information, and a flag to indicate whether or not to wait for an acknowledgement.

A *blocked send* is typically used when unreliable communications exist. For example, if the processors do not maintain message queues, a message will not be sent until the previous message has been received. A blocked send is used to avoid overwriting previous messages or message queues. A blocked send must be used if it is important that all messages be sent in a particular order (although this does not always guarantee that receiving processors receive messages in the order they were sent).

A *receive* is used to read a message sent from another processor. Receives can also be blocked or unblocked. The arguments in a receive statement typically include the message length, the message itself, and the sending processor's label. Other arguments often used include the type of message to be received and an indication of whether or not to send an acknowledgement.

A *blocked receive* is typically used when an algorithm requires a specific piece of data from another processor. The receiver then waits for the data to arrive. *Unblocked receives* are typically used in two ways. The most common use is to implement a global receive, in which a processor needs to receive messages on several input ports but the order is irrelevant. The other use is for asynchronous input. Here, the program continually checks the input port for a message. Depending on whether a message does or does not exist, a different task is done.

The programmer must be careful to avoid *deadlock* when programming message passing

systems. For example, when blocked receives are used, two processors might end up waiting for data from each other. If the programmer is not careful to match up sends and receives, problems will occur. For example, if processor p is waiting to receive a message (in blocked mode), but no processor sends a message to processor p, then it is suspended "forever."

5-2 Communication and Synchronization in Shared Memory Systems

Shared memory systems require an entirely different style of programming than distributed memory systems. There are several modes of operation, each requiring a different set of language extensions. Since data is shared, synchronization is needed to prevent out-of-sequence access to memory.

Atomic Operations

Atomic operations are operations which perform tasks with no possibility of interruption. They support multiprocess access to shared memory and are the building blocks for many essential multiprocessing synchronization mechanisms, such as locks, semaphores, and monitors (to be discussed). The most common situation requiring atomic operations is an attempt by two concurrent processes to change data in the same memory location. If process A reads memory location M after process B reads memory location M, but before process B writes the new value to memory location M, inaccuracies may occur if this situation was not accounted for in the algorithm. Atomic operations ensure that the reads and writes occur in proper sequence.

Serial Sections versus Critical Sections

A *serial section* is a section of code which is to be executed by one processor and skipped by all others. A *critical section* is a section of code which gets executed by all processors one at a time; that is, critical sections are parts of code in a set of processes that can not be executed in parallel. Figure 5-1 gives a procedure called *SumValues* (written in pseudo-code) which uses both a serial section and a critical section. This procedure is part of a program in which the main program distributes an array of values among the N processors of a parallel computer. Then *SumValues* is

called once for each processor with the processor's array of values and the number of values in the processor's array. Variable *Values* is declared as an array of real numbers. Variable *NumValues* is declared as an integer. Variable *Sum* is a global variable, defined in the main program, as a real number. Global variable *Sum* could have been initialized in the main program, but we wanted to illustrate a serial section to the reader.

```
PROCEDURE SumValues(Values, NumValues);

VAR
    Index: INTEGER;
    LocalSum: REAL;

BEGIN
    SERIAL SECTION
        Sum ← 0.0   {Sum is a global variable.}
    END SERIAL SECTION;
    LocalSum ← 0.0;
    FOR Index ← 1 TO NumValues
        LocalSum ← LocalSum + Values[Index];
    CRITICAL SECTION
        Sum ← Sum + LocalSum
    END CRITICAL SECTION
END
```

Figure 5-1. *N* processes call procedure *SumValues* with a different array of values (*Values*) and the number of values in their array (*NumValues*). Procedure *SumValues* sums up the values in *Values* into a local variable *LocalSum* and then adds that particular value to the global variable *Sum*. Since *Sum* is a global variable, we could get wrong results if all processors were allowed to initialize the variable, so we use a serial section. In particular, suppose the *N* processes are not synchronized when *SumValues* is called. Then if the serial section were removed, some process would initialize *Sum* to zero and start updating its value before other processes do the same thing, wiping out the previous partial sum and producing an incorrect solution. We need the critical section in the loop because *Sum* is a global variable. For example, if processor 1 and processor 2 each fetch the old value, add their respective contributions, and store the new value, the program will produce an incorrect result. The critical section guarantees that only one processor can update the global variable *Sum* at any instant.

Mutual Exclusion and Condition Synchronization

Two types of synchronization are commonly employed when using shared variables: *mutual exclusion* and *condition synchronization*. In systems with multiple concurrent processes, the presence of resources such as unit record peripherals and tape drives which must not be used simultaneously by several processes (if program operation is to be correct) introduces the requirement for exclusive access to these devices. This requirement may also be imposed on shared

objects such as a data segment during updating. Processes desiring exclusive access to a resource may compete for it. Mutual exclusion is the exclusiveness of access between processes.

Another situation occurs in a set of cooperating processes when a shared data object is in a state that is inappropriate for executing a given operation. Any process which attempts such an operation should be delayed until the state of the data object changes to the desired value as a result of other processes being executed. This type of synchronization is sometimes called condition synchronization.

Locks, Semaphores and Monitors

The *lock/unlock* mechanism is used to prevent out-of-sequence access to memory. For example, it can be used to ensure atomicity of the assignment statement ***Sum*** ← ***Sum*** + ***LocalSum*** from Figure 5-1, as follows:

```
Lock(Sum)
    Sum ← Sum + LocalSum
Unlock(Sum)
```

As explained above in the section on critical sections, such a mechanism is necessary for the following reason. If one process read the current value of *Sum* and performed the addition, resulting in a new value for *Sum*, but did not store this new result until after another process read the value of *Sum*, then the final answer would be incorrect. The assignment statement above is an example of a critical section.

Semaphores are used to ensure the mutual exclusion of access to shared resources by processes. E. W. Dijkstra invented the two operations **wait** and **signal**, which can be shared by many processes. The **wait** and **signal** operations are called primitives and are assumed indivisible. They operate on a special common variable called a *semaphore*, which indicates the number of processes attempting to use the critical section:

```
var s: semaphore;
```

where **semaphore** is defined over the nonnegative integers. Then the primitive **wait**(*s*) acts to

acquire permission to enter a critical section. The **signal**(*s*) primitive records the termination of a critical section. Figure 5-2 illustrates the use of a semaphore to implement mutual exclusion (discussed previously in this section).

Monitors are another concept to avoid conflicts when shared resources are used. All accesses must be done using special procedures. Specifically, a monitor consists of three items: (1) variables representing the state of some resource, (2) procedures that implement operations on that resource, and (3) initialization code. The initialization code initializes the values of variables before any procedure in the monitor is called. Monitor procedures look just like ordinary procedures in a programming language except the execution of procedures in the same module are guaranteed to be mutually exclusive. Thus, monitors are a structured way of implementing mutual exclusion.

```
wait(s):   s ← s - 1;
           IF s < 0 THEN
              BEGIN
              Block the process executing the wait(s) and put it in a
                 FIFO queue associated with the semaphore s.
              Resume the highest priority ready-to-run process.
              END

signal(s): s ← s + 1;
           IF s ≤ 0 THEN
              BEGIN
              If an inactive process associated with semaphore s
                 exists, then wake up the highest priority blocked
                 process associated with s and put it in a ready list.
              END
```

Figure 5-2. Using a semaphore to implement mutual exclusion.

Fork/Join and Single Program, Multiple Data (SPMD) Programming Styles

Two common programming styles used in shared memory systems are the *fork-join* style and the *single program, multiple data* (SPMD) style. In the SPMD style of programming, each processor is given the same program to run, but different code is executed depending on the data in shared memory or the processor id. The SPMD style of programming is also used in distributed memory systems.

In the fork-join style, a process uses the fork statement to start a new process while it continues executing, thus "forking" a single process into two processes. The invoking process can

synchronize with termination of the forked process by executing the join statement. That is, when the join statement is encountered by the invoking process, it must wait until the invoked process terminates before it can continue executing. For example, in Figure 5-3, a single process executes program **main** until the fork statement is encountered, at which time the execution of **main** continues and execution of **new** begins. If the process executing **main** reaches the join statement before the process executing program **new**, then the first process suspends execution. This process may continue execution of the statements after the join statement when the second process terminates execution of program **new**.

```
PROGRAM main;          PROGRAM new;
  ...                    ...
  fork new;              ...
  ...                    ...
  join new;              ...
  ...                  end;
```

Figure 5-3. Illustration of fork and join.

SPMD Programming Constructs

In the fork-join style of programming, the join statement itself is the means for synchronization. However, additional constructs are required for the SPMD style. One such construct is a barrier. A barrier is a point in the code where all processors must wait for the last processor to arrive. If the programmer is not careful, a processor could branch around a barrier, causing all of the other processors to wait "forever."

A second synchronization construct for SPMD programming is the wait until construct. Here, each processor continually checks a location in shared memory to see if some particular condition is met. The wait until construct is more flexible than the barrier in that processors can wait from different parts of a program for the condition to be met.

Figure 5-4 gives a code segment which uses both barrier and wait until constructs. This code segment (written in pseudo-code) illustrates the SPMD coding style on a shared memory system with N processors. Each processor executes the code asynchronously. Variable *Values* is declared

as an array of real numbers. Global variable *Sum* is declared as a real. Global variable *Synch* is declared as an integer. Variables *ProcessorID*, *StartingIndex*, and *NumValues* are declared as integers.

```
...
IF (ProcessorID = 0) THEN
   read data of known size into array Values;
   Sum ← 0.0;    {Sum is a global variable.}
   Synch ← 1;    {Synch is a global variable.}
ENDIF

Calculate StartingIndex (of the array) and NumValues according
  to N, the number of processes, and ProcessorID, the actual
  identification number of the process, so that the proper portion
  of array Values may be sent to procedure SumValues below.

WAIT UNTIL (Synch ≠ 0)
   CALL SumValues(Values(StartingIndex), NumValues);

BARRIER
   IF (ProcessorID = 0) THEN Print out Sum.
...
```

Figure 5-4. This code segment is an example of the SPMD coding style for a shared-memory system. Procedure *SumValues* is identical to the procedure in Figure 5-1. All processors but number 0 skip the code that reads the data. We use the wait until construct to synchronize the processors. Each processor continually checks global variable *Synch* until it takes on a different value from 0. While they are waiting for *Synch* to be set, all processors compute their own copies of the local variables *StartingIndex* and *NumValues*. Once *Synch* = 1, the waiting processors call *SumValues*. Next, we synchronize with a barrier to prevent processor 0 from writing *Sum* until all processors have finished adding their contributions. As soon as the last processor reaches the barrier, they all continue processing.

A third SPMD synchronization construct is the parallel do construct. Due to the relatively small expense of sharing data in shared memory systems, programmers often parallelize their code at the do loop level. That is, each processor is given a different iteration subset of the do loop (providing all iterations are independent). Two implementations of this distribution have been proposed, namely, the *self-scheduled* parallel do statement and the *prescheduled* parallel do statement. A self-scheduled parallel do works by giving the first value of the loop index to the first processor to arrive, the second index to the second processor to arrive, and so forth. When a processor completes the loop, another loop index is assigned to it, thus providing automatic load balancing. A prescheduled parallel do works by partitioning the loop ahead of time. Each

processor is allocated a set of loop indices, regardless of the time it takes each to finish.

CHAPTER 6

DESIGN OF ALGORITHMS

Introduction

The information in this chapter is based on [Karp 1987] and [Quinn 1987].

There are at least three ways to design a parallel algorithm to solve a problem: (1) one can detect and exploit any inherent parallelism in an existing sequential algorithm, (2) one can invent a new parallel algorithm, or (3) one can adapt another parallel algorithm that solves a similar problem. Each method has its place. Unless you are the very first person taking a look at a problem, someone else will more than likely have designed a sequential algorithm to solve it, so you may want to take advantage of that person's work rather than "reinvent the wheel." It may be possible to transform the sequential algorithm into parallel form, but it is not wise to blindly do so because a parallel algorithm made from a sequential algorithm having no obvious parallelization usually exhibits poor speedup. In many cases, the architecture itself demands that a new approach be taken, so one might be better off starting from scratch or exploring related parallel algorithms for the same area or related architecture.

Insight plays an important role to the parallel algorithm designer. If a well-known sequential algorithm already exists, the designer may wish to use this sequential algorithm as a starting point for writing the parallel version. The designer will have to apply some external knowledge of the problem in order to break it up if the sequential algorithm is not particularly parallelizable. For example, consider the simple problem of summing n integer values, for $n > 0$. A sequential

algorithm may look something like the following:

```
BEGIN
    Sum ← A_FirstIndex
    FOR Index ← SecondIndex TO LastIndex DO
        Sum ← Sum + A_Index
    ENDFOR
END
```

For $n = 4$, the additions would be done in the following order:

$$((A_1 + A_2) + A_3) + A_4$$

The parallel algorithm designer would have to ask himself/herself whether this is an inherently sequential process. Examining the way the parentheses are grouped, it seems that A_1 and A_2 must be summed before adding A_3 to the subtotal, and only after this subtotal is made can A_4 be added. However, we have some external knowledge .hat addition is an associative operation and, therefore, addition can be done in parallel. The expression can be rewritten as follows:

$$(A_1 + A_2) + (A_3 + A_4)$$

Seeing the expression in this form makes it clear that A_1 and A_2 may be added at the same time as A_3 and A_4 are added; that is, in parallel.

6-1 Designing Algorithms for SIMD Computers

Recall from Chapter 1 that the n processing elements (PEs) of an SIMD computer execute the same instruction at the same same time on the contents of their ow. .ocal memory. A control unit stores the program and broadcasts instructions to all n PEs simultaneously. The PEs are linked via some interconnection network.

Since the PEs of an SIMD computer perform operations in lockstep, the algorithm designer need not worry about synchronizing the processors. The designer does, however, have to be concerned with minimizing the amount of communication because communication costs can be

expensive.

We now give three algorithms for summing n values on the SIMD model with (1) a hypercube interconnection network, (2) a perfect-shuffle interconnection network, and (3) a mesh interconnection network. A double arrow ($\Leftarrow$) in an algorithm represents the communication of data from one node to a neighboring node. A single arrow ($\leftarrow$) represents an assignment operator.

Finding the Sum of n Values on the SIMD Hypercube Model

The following algorithm finds the sum of n values on an SIMD computer with a hypercube interconnection network. An explanation of the algorithm follows the code.

```
0. BEGIN
1.    FOR Iteration ← log(n) - 1 DOWNTO 0 DO
2.       ActiveNodes ← 2^Iteration
3.       FOR ALL P_NodeLabel, where 0 ≤ NodeLabel < ActiveNodes DO
4.          ReceivedValue_NodeLabel ⇐ ValueAtNode_NodeLabel + ActiveNodes
5.          ValueAtNode_NodeLabel ← ValueAtNode_NodeLabel + ReceivedValue_NodeLabel
6.       ENDFOR
7.    ENDFOR
8. END
```

Explanation. Line 1 sets up a for loop to iterate log(n) times, where n is the number of processors. Notice that the n processors are labeled 0, ..., n - 1 with log(n) bits each. The for all loop in line 3 activates specific processors to execute the statements before the matching endfor **in parallel, in lockstep** fashion. Line 2 initializes *ActiveNodes*, which indicates how many processors will be active in the for all loop. Recall that the double arrow ($\Leftarrow$) represents the communication of data from one node to a neighboring node. Thus line 4 represents the communication of the value at node *NodeLabel* + *ActiveNodes* to node *NodeLabel*. Figure 6-1 illustrates this algorithm for n = 16.

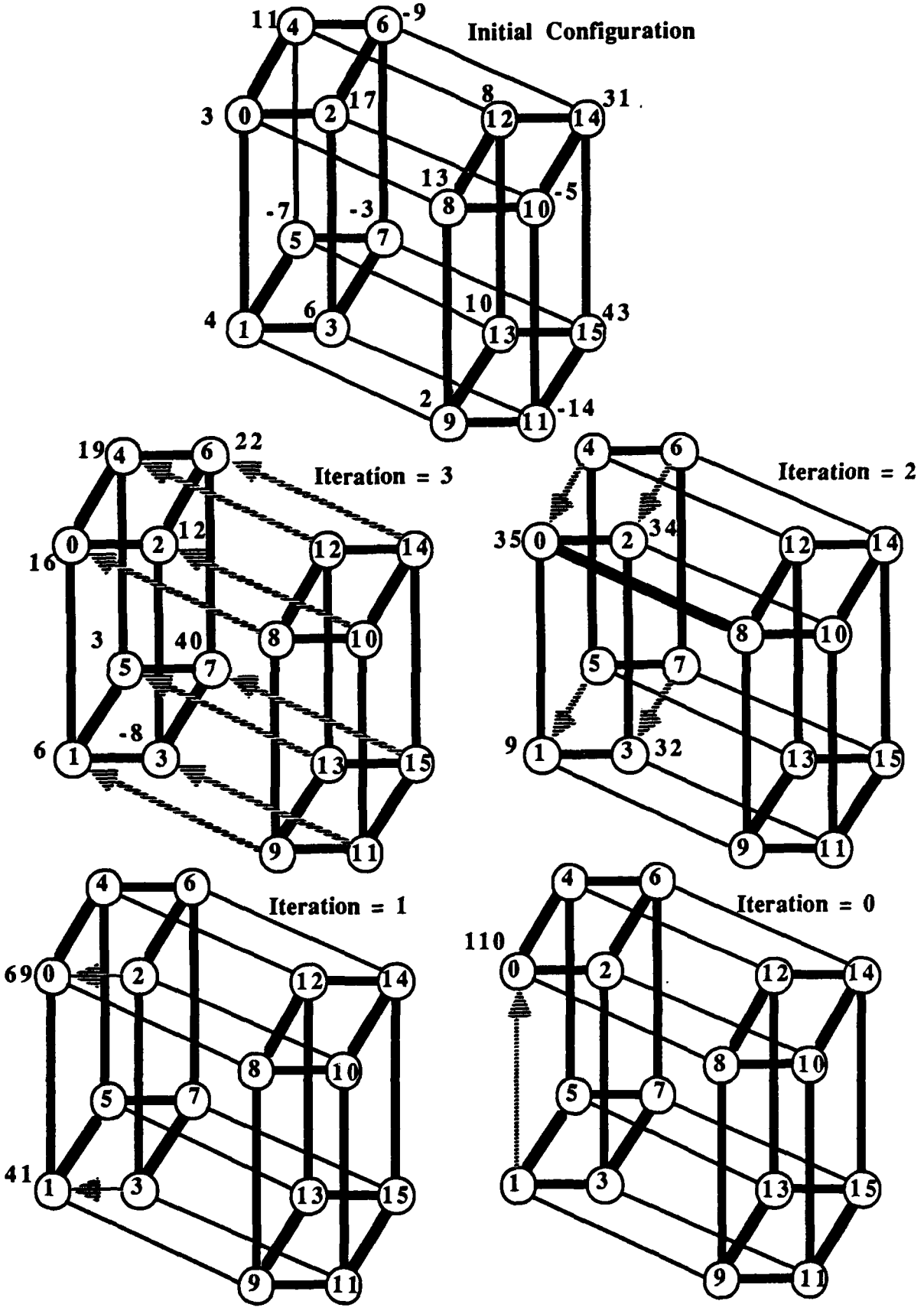


Figure 6-1. A trace of the SIMD hypercube algorithm which sums n values. Here, $n = 16$.

Finding the Sum of n Values on the SIMD Shuffle-Exchange Model

The following algorithm finds the sum of n values on an SIMD computer with a shuffle-exchange interconnection network. An explanation of the algorithm follows the code.

```
0. BEGIN
1.    FOR Iteration ← 1 TO log(n) do
2.       FOR ALL P_NodeLabel, where 0 ≤ NodeLabel < n do
3.          PerfectShuffle(PartialSums_NodeLabel)
4.          Copy_NodeLabel ← PartialSums_NodeLabel
5.          Exchange(Copy_NodeLabel)
6.          PartialSums_NodeLabel ← PartialSums_NodeLabel + Copy_NodeLabel
7.       ENDFOR
8.    ENDFOR
9. END
```

Explanation. Line 1 sets up a for loop to iterate log(n) times, where n is the number of processors. The for all loop in line 2 activates specific processors to execute the statements before the matching endfor **in parallel, in lockstep** fashion. The *PerfectShuffle* operation used in line 3 and the *Exchange* operation used in line 5 are illustrated in Figure 6-2. Within the for all loop, the partial sums are perfectly shuffled, a copy of these shuffled partial sums is made, the exchange operation is carried out on the copy, and each "shuffled" partial sum is added to its corresponding "exchanged" partial sum. See Figures 6-3 and 6-4 for a trace of this algorithm and an illustration of the trace for $n = 16$, respectively.

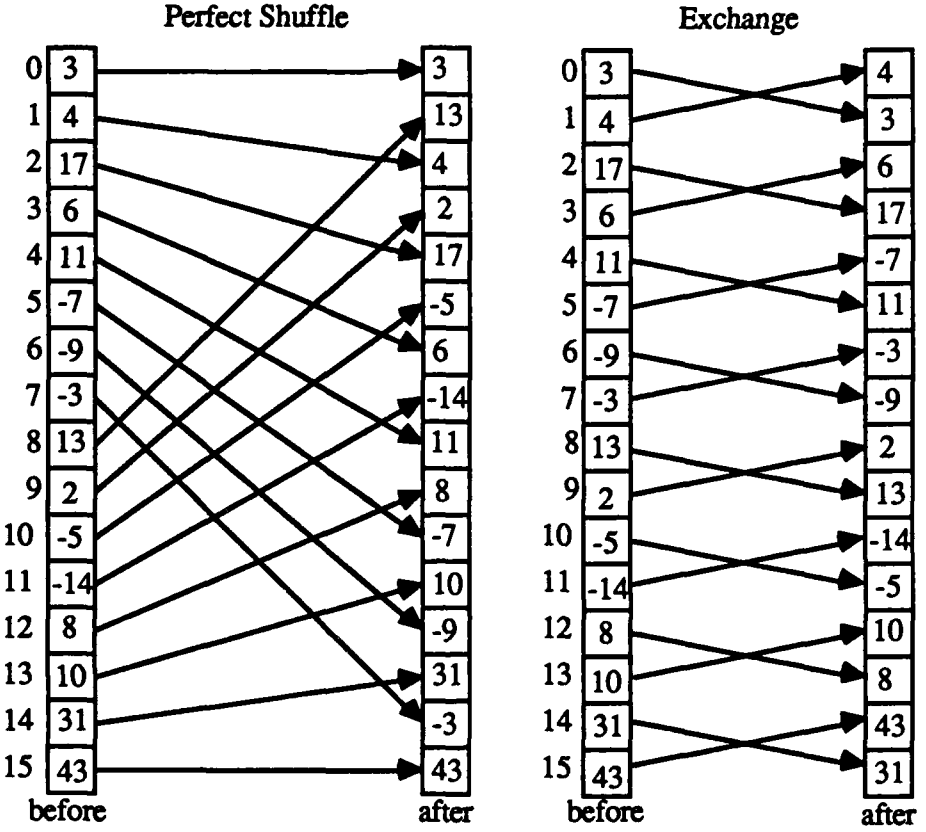


Figure 6-2. Operations shuffle and exchange.

	Iteration 1				Iteration 2			Iteration 3			Iteration 4		
	initial values	shuffled initial values	exchanged copy of shuffled values	partial sums	shuffled partial sums	exchanged copy of shuffled values	partial sums	shuffled partial sums	exchanged copy of shuffled values	partial sums	shuffled partial sums	exchanged copy of shuffled values	final sum
0	3	3	13	16	16	19	35	35	34	69	69	41	**110**
1	4	13	3										
2	17	4	2	6	19	16							
3	6	2	4										
4	11	17	-5	12	6	3	9	34	35				
5	-7	-5	17										
6	-9	6	-14	-8	3	6							
7	-3	-14	6										
8	13	11	8	19	12	22	34	9	32	41	41	69	
9	2	8	11										
10	-5	-7	10	3	22	12							
11	-14	10	-7										
12	8	-9	31	22	-8	40	32	32	9				
13	10	31	-9										
14	31	-3	43	40	40	-8							
15	43	43	-3										

Figure 6-3. A trace of the SIMD shuffle-exchange algorithm which sums n values. Here, $n = 16$. The solid horizontal line midway down the table is a reference point to help the reader follow the trace.

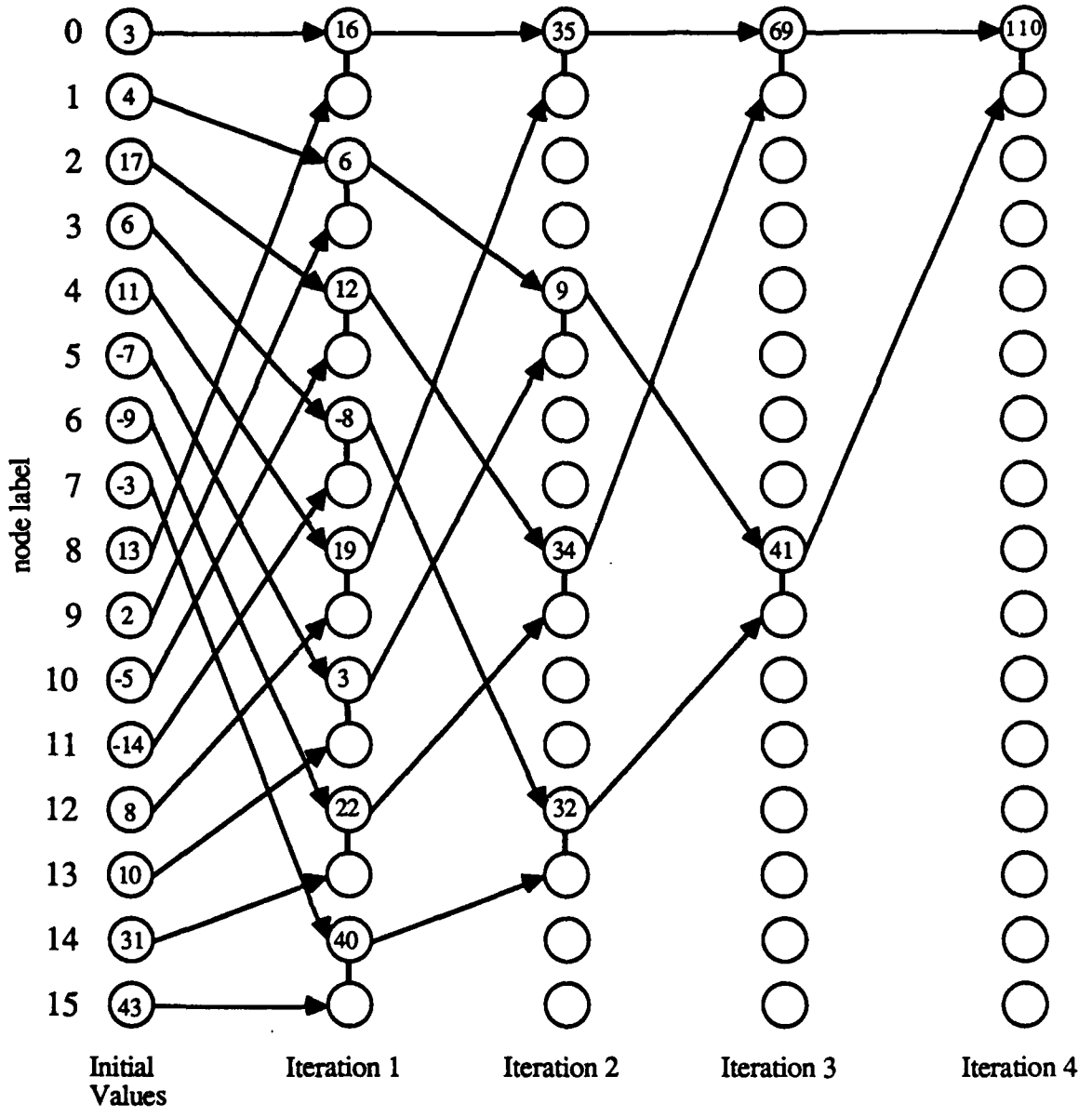


Figure 6-4. An illustration of the SIMD shuffle-exchange algorithm trace from Figure 6-3.

Finding the Sum of n Values on the SIMD Mesh Model

The following algorithm finds the sum of n values on an SIMD computer with a mesh interconnection network. An explanation of the algorithm follows the code.

0. BEGIN
1. FOR $Column \leftarrow NumberOfCols$ - 1 DOWNTO 1 DO
2. FOR ALL $P_{Row, Column}$, where $1 \leq Row \leq NumberOfRows$ DO
3. $ReceivedValue_{Row, Column} \Leftarrow ValueAtNode_{Row, Column+1}$
4. $ValueAtNode_{Row, Column} \leftarrow ValueAtNode_{Row, Column} + ReceivedValue_{Row, Column}$
5. ENDFOR
6. ENDFOR
7. FOR $Row \leftarrow NumberOfRows$ - 1 DOWNTO 1 DO
8. FOR ALL $P_{Row, 1}$ DO
9. $ReceivedValue_{Row, 1} \Leftarrow ValueAtNode_{Row+1, 1}$
10. $ValueAtNode_{Row, 1} \leftarrow ValueAtNode_{Row, 1} + ReceivedValue_{Row, 1}$
11. ENDFOR
12. ENDFOR
13. END

Explanation. Line 1 sets up a for loop to iterate *NumberOfCols* - 1 times (we assume the number of rows equals the number of columns, and n = number of nodes = (number of rows)^2). The for all loop in line 2 activates specific columns of processors to execute the statements before the matching **endfor in parallel, in lockstep** fashion. The following is repeated *NumberOfCols* - 1 times: every node in some specified column simultaneously sends its value west, these values are then simultaneously added to the values at the nodes they were sent to. The for all loop in line 8 activates specific nodes in column 1 to execute the statements before the matching **endfor in parallel, in lockstep**. This time, values are sent north. The final sum ends up in $node_{1, 1}$. Figure 6-5 gives an illustration of the algorithm for $n = 16$.

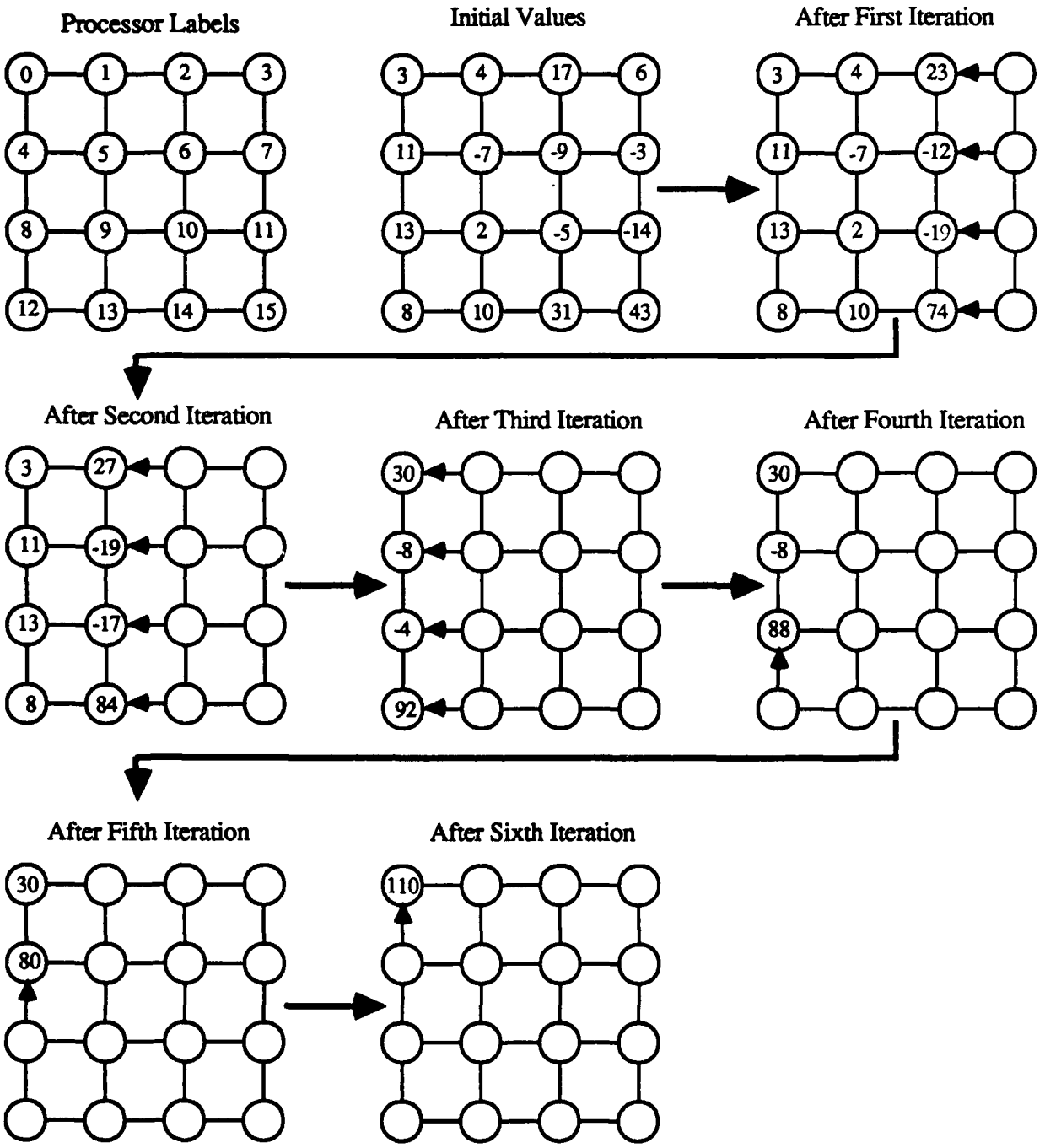


Figure 6-5. A trace of the SIMD mesh algorithm which finds the sum of n values. Here, $n = 16$. Note that rows and columns are numbered starting from 1, not 0.

6-2 Designing Algorithms for MIMD Computers

Recall from Chapter 1 that an MIMD computer typically consists of n processing elements (PEs), n memory modules, and an interconnection network. Each of the PEs stores and executes its own program and fetches its own data on which to operate. The interconnection network provides communications among the processors and memory modules. MIMD machines can be organized as shared memory or distributed memory machines.

The goal of the designer of algorithms for MIMD computers is the same as that of SIMD computers: given a problem with a certain amount of inherent parallelism and a number of processors, find an algorithm that best utilizes these processors to solve the problem as quickly as possible.

Classifying MIMD Algorithms

MIMD algorithms can be divided into three categories: (1) pipelined algorithms, (2) partitioned algorithms, and (3) relaxed algorithms. A *pipelined algorithm* is an ordered set of stages in which the output of one stage is the input to the next stage. The input to the algorithm is the first stage's input and the output of the last stage is the output of the algorithm. All stages must produce results at the same rate, or else the slowest stage will become a bottleneck. An example of a pipelined MIMD algorithm is a parallel compiler with individual stages - scanning, parsing, code generation, and code optimization - assigned to a set of processors.

In a *partitioned algorithm*, processors share a computation. A problem is divided into a number of subproblems to be solved by individual processors. Through synchronization among the processors, the solutions of the subproblems are combined to form the problem solution. Partitioned algorithms are sometimes called *synchronized algorithms* for this reason. Figure 6-6 illustrates a partitioned algorithm to find the sum of n values on the MIMD model with a hypercube interconnection. An explanation of this algorithm is as follows: The host computer executes a different program than the nodes. The host's program distributes the array of values to be summed among the nodes; a different part of the array is sent to each of the nodes labeled 0 to n - 1 using the **send** function (see Section 4-1). Next, the host executes a **receive** and waits for the final sum to

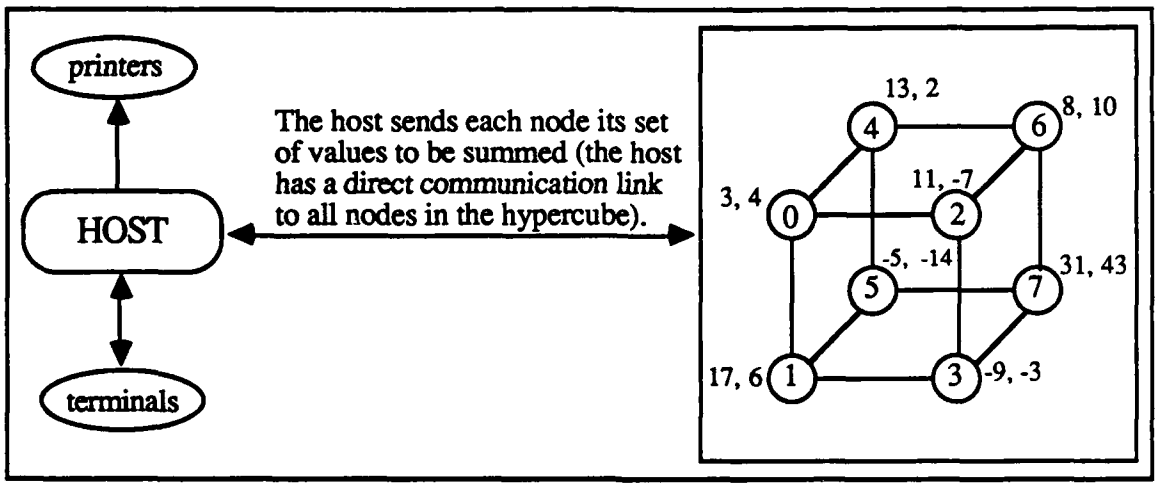
arrive. The node program does the following. Each node **receives** its set of values from the host and sums them up. These n partial sums are then combined into the final sum as follows. In a loop starting with the highest dimension, each node in dimension d sends its partial sum to its neighboring node in dimension d - 1. Each node in dimension d - 1 **receives** a partial sum from its neighboring node in dimension d and adds the received partial sum to its own. Eventually, node 0 will contain the final sum. Node 0 sends this sum to the host computer.

Notice that the partitioned algorithm of Figure 6-6 resembles the SIMD hypercube algorithm of Figure 6-1, the main difference being that the processors in the MIMD computer operate asynchronously while the processors in the SIMD computer operate synchronously. Figure 6-6 may be misleading in this respect: the combine phase has three stages, each stage marked by arrows showing flow of data. Although the arrows make it look like the data is being sent at the same time, this is not the case. For example, in the first stage of the combine phase, node 4 is sending data to node 0, node 5 is sending data to node 1, node 6 is sending data to node 2, and node 7 is sending data to node 3 *asynchronously*. These four activities are **not** occurring in lock step.

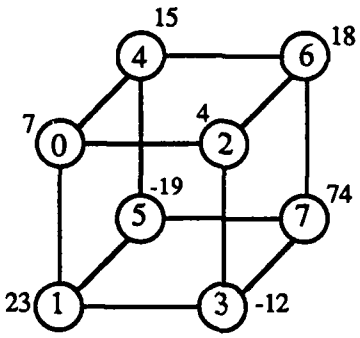
Partitioned algorithms can be further sudivided into *prescheduled* and *self-scheduled* algorithms. In a prescheduled algorithm, each process is allocated its share of the computation at compile time. In a self-scheduled algorithm, the work is not assigned to the process until run time. A global list of work to be done is kept, and when a process is without work, another task is removed from the list. Processes schedule themselves as the program executes.

In a *relaxed algorithm*, no process synchronization exists. In other words, no processor ever has to wait for another processor to provide data. Instead, processors work with the most recently available data. Sometimes these algorithms are called *asynchronous algorithms*.

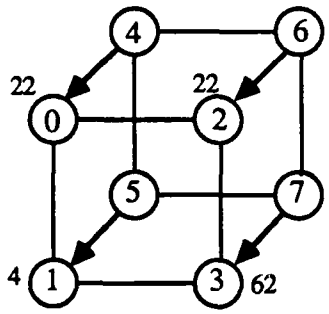
Note that these three methods of algorithm design for MIMD computers are not mutually exclusive. Often, an algorithm is designed with features from all three types. For example, at the highest level, an algorithm may be divided into stages forming a pipeline. One or more of these stages may be parallelized further through partitioning, while other stages may be parallelized through relaxation.



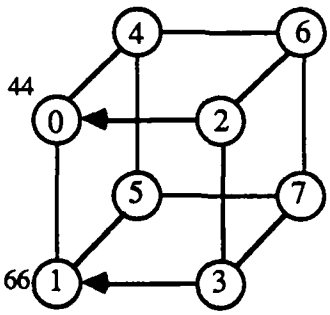
Each node sums its own values:



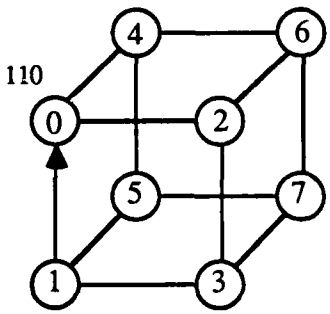
Combine Phase:



Each node in the third dimension sends its partial sum to its neighboring node in the second dimension. Each node in the second dimension receives the partial sum from the third dimension and adds it to its own.



Each node in the second dimension sends its partial sum to its neighboring node in the first dimension. Each node in the first dimension receives the partial sum from the second dimension and adds it to its own.



Node 1 sends its partial sum to node 0. Node 0 receives the partial sum from node 1 and adds it to its own. (Node 0 sends the final sum to the host.)

Figure 6-6. A partitioned algorithm to find the sum of n values on the MIMD hypercube model.

Finding the Sum of *n* Values on a Shared Memory, MIMD Computer

The following algorithm finds the sum of *n* values on a shared memory, MIMD computer using the lock/unlock synchronization mechanism (discussed in Chapter 5). An explanation of the algorithm follows the code.

```
0.  BEGIN
1.      GlobalSum ← 0
2.      FOR ALL P_NodeLabel, where 0 ≤ NodeLabel < NumNodes DO
3.          LocalSum ← 0
4.          FOR Label ← NodeLabel TO NumValues STEP NumNodes DO
5.              LocalSum ← LocalSum + Value_Label
6.          ENDFOR
7.          Lock(GlobalSum)
8.              GlobalSum ← GlobalSum + LocalSum
9.          Unlock(GlobalSum)
10.     ENDFOR
11. END
```

Explanation. There are p processors labeled $P_0, P_1, \ldots, P_{p-1}$ and n variables labeled $Value_0$, $Value_1, \ldots, Value_{n-1}$, stored in global memory containing the values to be added. A global variable called *GlobalSum* is used to store the total sum. Each processor has two local variables: (1) *Label*, which is used as the for loop index in line 4 and acts as a variable subscript, and (2) *LocalSum*, which contains the processor's partial sum. The for all loop in line 2 activates processors to **asynchronously** execute the statements before the matching endfor in line 10 (once all processors reach the endfor, a single processor resumes execution with the next statement after the endfor). Inside the for all loop, each processor initializes its local sum variable, *LocalSum*, to 0. Then each processor enters the for loop in lines 4 through 6 which sums up the processor's values into *LocalSum*. Lines 7 through 9 implement a critical section: Line 7 locks the global variable

GlobalSum so that only the node locking the variable can read from it or write to it. A variable can be locked by one processor at a time and only the process that locks the variable may unlock it. Line 8 alters the value of *GlobalSum* (i.e., the processor which locked *GlobalSum* adds its local sum, *LocalSum*, to *GlobalSum*) and line 9 unlocks the variable. To give the reader a better understanding of this algorithm, Figure 6-7 shows a trace for $n = 16$.

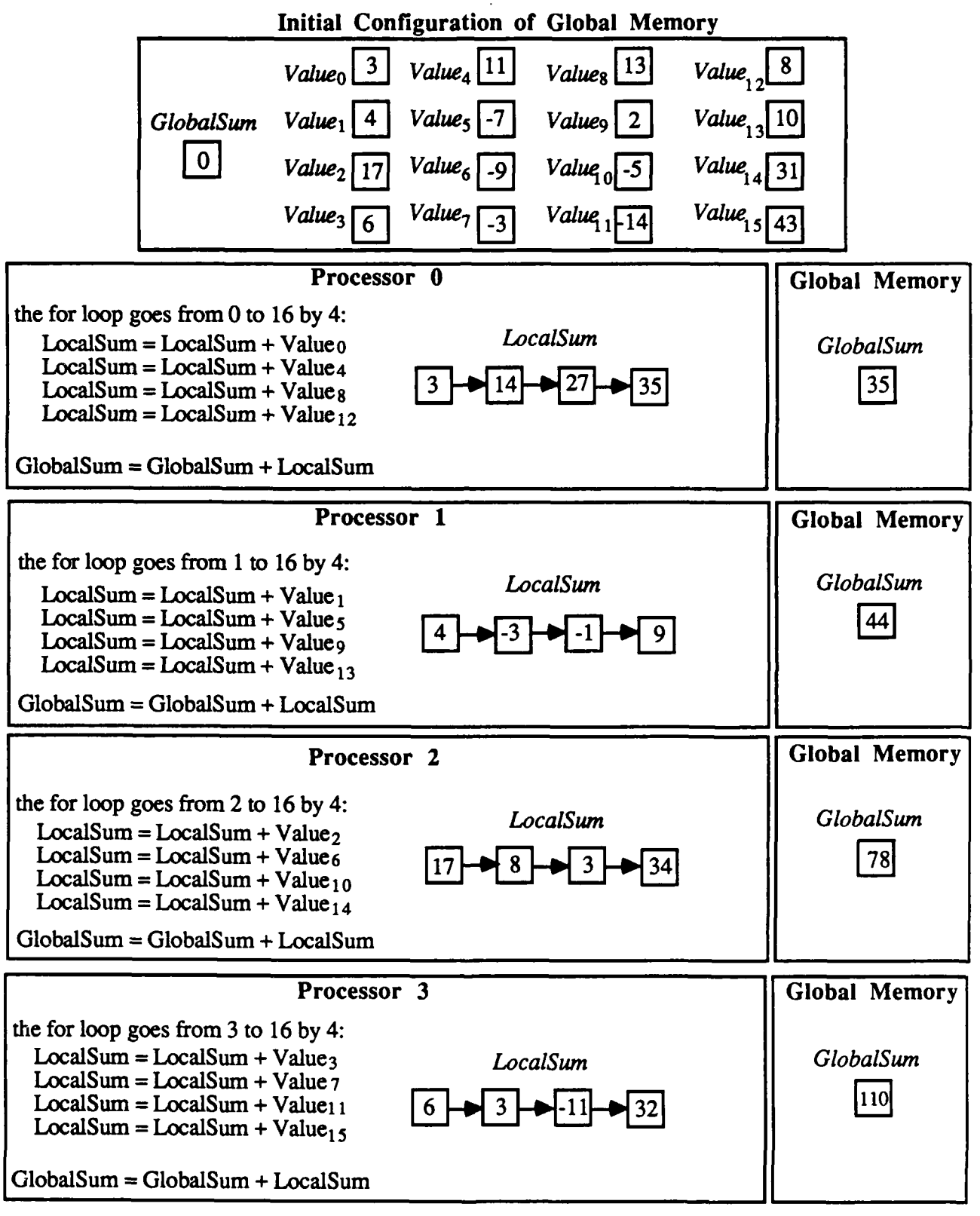


Figure 6-7. A trace of the MIMD shared memory algorithm which sums *n* values.

CHAPTER 7

OPERATING SYSTEMS

Introduction

The information in this chapter is based on [Hwang Briggs 1984], [Trakhtengerts Shuraits 1982] and [Russell Waterman 1987].

In this section, we discuss operating system issues for parallel computers. A classification of parallel computer operating systems is given first. We then discuss the familiar UNIX operating system as a basis for parallel computer operating systems, and we discuss a new operating system called *Mach*.

THE CONTENTS OF THIS CHAPTER MAY CHANGE WITH THE NEXT EDITION OF THIS DOCUMENT, DUE TO BE RELEASED IN LATE 1988.

7-1 Requirements for Parallel Computer Operating Systems

The operating system requirements of a parallel computer are quite similar to the operating system requirements of a large computer system utilizing multiprogramming (a multiprogramming operating system allows more than one program to be in some state of execution at the same time). However, when multiple processors must work simultaneously and the operating system must support multiple asynchronous tasks which execute concurrently, the operating system is more complex.

The usual functional requirements of an operating system for a multiprogrammed computer include resource allocation and management schemes, memory and dataset protection, prevention

of system deadlocks, and abnormal process termination (exception handling). In addition to these capabilities, parallel computer operating systems also require techniques for efficient utilization of resources, and, thus, must provide I/O and processor load-balancing schemes. They must also be capable of providing system reconfiguration schemes to support graceful degradation in the event of a failure.

7-2 Classification of Parallel Computer Operating Systems

There are basically three organizations that have been utilized in the design of operating systems for parallel computers: (1) master-slave configuration, (2) separate supervisor for each processor, and (3) floating supervisor control. In the *master-slave configuration*, one processor, called the master, maintains the status of all processors in the system and apportions the work to all the slave processors. In the *separate supervisor system*, a separate kernel (the basic subset of an operating system) runs in each processor, and each processor services its own needs. The *floating supervisor control* scheme treats all the processors as well as other resources as an anonymous pool of resources. The floating supervisor control scheme is the most difficult and the most flexible mode of operation. The supervisor routine floats from one processor to another, although several of the processors may be executing supervisory service routines simultaneously.

[Hwang Briggs 1984] provides a table of the major characteristics, advantages, and disadvantages of the above three types of operating systems for parallel computers. This table is illustrated in Figure 7-1.

7-3 UNIX and Mach Operating Systems

The idea of adapting an existing operating system for parallel computer use is a natural one. Thus, the majority of commercial and university endeavors have chosen either to build upon a version of the UNIX operating system or to expand upon a new operating system called *Mach*, which is introduced below.

UNIX Operating System

Ever since Brian W. Kernighan and Dennis M. Ritchie wrote UNIX 18 years ago, designers have continually adapted it to new, more powerful computer architectures. UNIX is especially popular in academic settings and research laboratories. It is no surprise then that the majority of non-von Neumann computers use a UNIX-based operating system. Many computer scientists and engineers are familiar with UNIX, and therefore, they are pleased when vendors provide UNIX-based operating systems for their new parallel computers.

UNIX was originally written as a portable, general-purpose, time-sharing operating system, and it was developed to run on a *single* processor. Therefore, when a UNIX-based operating system is implemented for a parallel computer with the goal of maintaining some degree of UNIX compatibility, many challenges arise. For example, the operating system must distinguish between implementations with shared memories or distributed memories, any number of communication (interconnection) networks, and message passing or shared-memory programming styles.

To meet these challenges, many companies and universities have written their own parallel processing operating systems, each very different, but each based on the original UNIX philosophies.

Mach Operating System

The University of California at Berkeley eventually added many enhancements to the original BSD UNIX under sponsorship of the Defense Advanced Research Projects Agency (DARPA). Although many of these enhancements were in response to emerging distributed processing needs, the operating system did not really address multiple-processor-specific issues. Instead, the kernel ended up being far more complicated than the original version. Deciding it was necessary to return to simplicity, DARPA recently funded researchers at Carnegie-Mellon University to undertake the development of a new operating system called *Mach*.

Mach provides many new features specifically aimed at parallel processing systems. It supports four basic programmer-visible abstractions: *port*, *task*, *thread*, and *message* (discussed below). Mach provides large, sparsely populated demand-paged address spaces (a mechanism for retrieving

data on demand), an interprocessor communications facility based on message passing, and a remote procedure call capability for interfacing with tasks written in C, Lisp, Ada, and Pascal. Mach supports the goal of running one operating system on many different classes of machines in a variety of configurations, all with a consistent user interface.

The first of the four fundamental abstractions supported by Mach is a *port*. A Mach port is simply a queue for messages that is protected by the kernel. All traffic within Mach makes references to ports as read or write destinations, using the primitives send and receive.

The second and third abstractions arise from separating the traditional notion of a process into two subconcepts. *Tasks* contain the resources associated with a process (e.g., the address space, file descriptors, and port-access capabilities). They do not perform computations themselves, but serve as a framework in which *threads* can operate. A thread is the control unit most basic to CPU utilization, containing the minimal processing state associated with a computation: a program counter, a stack pointer, and other hardware register state information.

A Mach task may contain multiple threads, but each thread is associated with exactly one task. Since each thread may access all of its associated task's resources, including shared memory, the Mach design naturally supports parallel-programming techniques.

A *message* consists of a fixed-length header and a typed collection of data objects used in communications between threads. Messages come in all sizes and may contain port-access capabilities in addition to data. Operations on objects other than messages are performed by sending messages to ports that are designed to represent them. By implementing message passing as well as shared-memory techniques, Mach overcomes many of the UNIX limitations.

Master-slave operating system	Separate supervisor in each processor	Floating-supervisor operating system
• The executive routine is always executed in the same processor. If the slave needs service that must be provided by the supervisor, then it must request that service and wait until the current program on the master process is interrupted and the supervisor is dispatched. The supervisor and the routines that it uses do not have to be reentrant (as opposed to replicated) since there is only one processor using them. • Having a single processor executing the supervisor simplifies the table conflict and lock-out problem for control tables. The overall system is comparatively inflexible. This type of system requires comparatively simple software and hardware. • The entire system is subject to catastrophic failures that require operator intervention to restart when the processor designated as the master has a failure or irrecoverable error. • Idle time on the slave system can build up and become quite appreciable if the master cannot execute the dispatching routines fast enough to keep the slave(s) busy. • This type of operating system is most effective for special applications where the work load is well defined or for asymmetrical systems in which the slaves have less capability than the master process.	• Each processor services its own needs. In effect, each processor (supervisor) has its own set of I/O equipment, files, etc. • It is necessary for some of the supervisory code to be reentrant or replicated to provide separate copies for each processor. • Each processor (actually each supervisor) has its own set of private tables, although some tables must be common to the entire system, and that creates table access control problems. • The separate supervisor operating system is as sensitive as is the master-slave system; however, the restart of an individual processor that has failed will probably be quite difficult. • Because of the point immediately above, the reconfiguration of I/O usually requires manual intervention and possibly manual switching.	• The "master" floats from one processor to another, although several of the processors may be executing supervisor service routines at the same time. • This type of system can attain better load balancing over all types of resources. • Conflicts on service requests are resolved by priorities that can be set statically or under dynamic control. • Most of the supervisory code must be reentrant (as opposed to replicated) since several processors can execute the same service routine at the same time. • Table access conflicts and table lock-out delays can occur, but there is no way to avoid this with multiple supervisors; the important point is that they must be controlled in such a way that system integrity is protected.

Figure 7-1. Operating system configurations for a parallel computer. (*Reentrant code* is non-self-modifying code. If code is reentrant, then it never changes during execution. Thus two or more processes can execute the same code at the same time.)

CHAPTER 8

SOFTWARE TOOLS

Introduction

In this chapter, we outline an ideal parallel programming environment, in which users are aided with system tools and visual aids for program trace, resource mapping, and analysis of data structures. Then we provide a sampling of actual software tools available for parallel computers.

THE CONTENTS OF THIS CHAPTER MAY CHANGE WITH THE NEXT EDITION OF THIS DOCUMENT, DUE TO BE RELEASED IN LATE 1988.

8-1 Programming Environments

The following information is based on [Hwang 1987].

A *programming environment* is a collection of tools that can be used to develop software. New issues in programming environment design have arisen with the advent of parallel architectures. Three key issues which must be addressed are (1) What information is going to be collected? (2) How is the information recorded and displayed? and (3) What mechanisms should be provided to alter execution of a program? The performance of all functional units should be able to be monitored by the programmer.

A programmer's ability to understand what is going on with his/her parallel program is greatly improved when all of the traditional views of a program are available to be instantly viewed on the screen. These traditional program views include the program listing, data type schema, the symbol table, the flow graph, the execution stack, and I/O dialogue.

The hardware of an *ideal* programming environment for parallel algorithms designers consists of a parallel computer coupled to a high resolution, color display, a mouse, a keyboard, and a videodisk recorder and player. The software contains support for the color graphics plus associated support for windowing. There must be software for manipulating the videodisk, both for recording and playback. An ideal programming environment should also be user friendly.

8-2 Actual Software Tools

We now discuss a variety of software tools which have been designed for parallel computers. These include debuggers, performance monitors, software directories, and programming environments. While there are probably hundreds of tools available for today's parallel computers, we have provided a select number of typifying examples.

A Knowledge-Based Parallelization Tool

[Brandes Sommer 1987] describes a parallelization tool which supports the development of software for parallel computers by helping users interactively by detecting 'bottlenecks' in the computation and by suggesting and making transformations which increase the potential of parallelism. The tool can be used for most high level languages (also parallel ones) as well as for many different kinds of parallel hardware (unspecified in [Brandes Sommer 1987]). In addition, this parallelization tool already supports software development - it is embedded in a modern programming environment where all tools are integrated (the specific environment is not given a name in [Brandes Sommer 1987], but the reference [Nagl 1985] is given).

Belvedere

Belvedere is a "pattern-oriented" debugger designed to investigate the animation and manipulation of interprocess communication patterns for nonshared-memory, message passing architectures. Belvedere is a trace-based, post-mortem debugger. It provides animations of program behavior both in terms of primitive simulator events and in terms of abstract events from user-defined perspectives. Belvedere assists only with parallel programming errors; it does not provide any aid

in detecting errors in the sequential sections of code. [Hough Cuny 1987]

Cray Directory of Supercomputer Applications Software

The following information is based on [Supercomputer 1987].

The Cray Directory of Supercomputer Applications Software is published by Cray Research, Inc., and it is updated every six months. The January 1987 issue contained on the order of 400 software entries. Two-thirds of the entries are vendor supplied and supported, and one-third come from universities and public domain suppliers.

Each Directory entry represents sofware that runs on a Cray supercomputer and is available to the requester. An entry consists of a description of the software and information about how to obtain the software, as well as find out more about it. The software listed satisifies computational needs in many science and engineering categories, as well as applications in finance and transportation industries.

The Cray Directory of Supercomputer Applications Software is available to Cray customers, potential customers, and organizations or individuals that provide services to the same. Additional information can be obtained from Trudy Sprague, Applications Information Analyst, Cray Research, Inc., 1333 Northland Drive, Mendota Heights, Minnesota 55120.

Faust

Faust is a software engineering environment for scientific computing being developed at the Center for Supercomputing Research and Development in Illinois that is targeted at integrating several software development tools through a common window-based interface. For example, a user wanting to develop an application at the source-code level may bring up a textual window and enter Fortran source using a conventional text editor, as illustrated in Figure 8-1a. If the user would rather see the application at a higher level, an "unzoom" function can be invoked to bring up the corresponding subroutine interconnection graph as illustrated in Figure 8-1b. Faust can automatically create the subroutine call graph (if it does not already exist) from source code. However, if desired, the user may do the original editing at the graphic level of abstraction and

associate source code for each "block" as the implementation proceeds. Faust also supports other levels of detail including process graphs that represent parallelism as well as data dependence graphs for aiding interactive restructuring. [Padua Guarna Lawrie 1987]

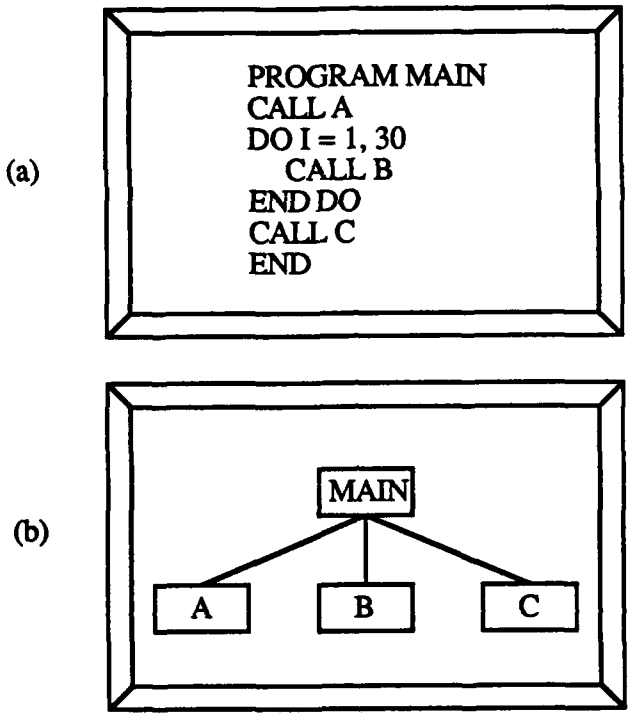


Figure 8-1. (a) simple application being examined at the source code level. (b) same application at the subroutine interconnect level.

Instant Replay

The following information is based on [Padua Guarna Lawrie 1987].

The most challenging aspect of parallel debugging is the timing conflicts introduced by interacting, independently running processors. The series of states through which a *serial* program passes is not time dependent and is therefore repeatable, providing the opportunity for an unlimited number of reruns in order to localize run-time anomalies. On the other hand, the set of states through which a *parallel* program passes is dynamic (i.e. changing) and very sensitive to the speed at which each processor is progressing. For this reason, program errors may surface infrequently.

Instant Replay, developed at the University of Rochester, is a debugging environment targeted

at helping users debug parallel programs on the BBN Butterfly (Section 3-1). Instant Replay attacks the repeatability problem by regulating and recording access to shared data objects. By introducing some small run-time overheads, Instant Replay attaches aging information to all shared objects and records revision numbers as these objects are updated and diseminated. In addition to recording this revision information, the run-time system has the ability to "replay" the application while insuring the same access sequences to shared objects. This gives the programmer the capability to perform the cyclic rerunning necessary to do incremental debugging on a parallel machine.

Monit

Monit is a performance monitoring tool for parallel programs. To date, it has been used with a new parallel programming language called PPL [Schwetman 1986]. PPL is a superset of C; it has features which allow users to easily write parallel programs which are realized as concurrently executing processes. PPL programs currently execute on a Sequent multiprocessor system. A performance monitoring facility within PPL creates a file of interesting events during program execution. Monit, an interactive program for a SUN-3 workstation, processes this event file to produce statistical summaries and time-based bar graphs, thereby giving the user new insight into the performance characteristics and behavior of the program. [Kerola Schwetman 1987]

Parafrase and Parafrase II

The following information is based on [CSRD 1987].

Parafrase is a program restructuring tool which performs source-level transformations on Fortran code, with the object of producing code that takes advantage of the parallelism available in various computer architectures. Parafrase is capable of restructuring Fortran programs in a vector or in a parallel form suitable for execution in pipeline/array and parallel computers respectively. Although the output of Parafrase is not executable, it is useful in analyzing and evaluating parallal programs. In addition, Parafrase can be used to estimate the performance of the resulting code on the target machine. It provides measurements for each program analyzed that help in the estimation

of the expected speedup due to restructuring as well as a number of other performance indices.

Parafrase II is being developed as a more powerful restructuring compiler that will produce executable code. It will be a multilanguage compiler allowing restructuring of both Fortran and C. In addition, Parafrase II will have the ability to perform overhead analysis and do prescheduling. This research and development effort is going on at the Center for Supercomputing Research and Development.

Parallel Fortran Converter (PFC)

PFC translates sequential programs written in Fortran to Fortran 8x, the Fortran standard proposed by the Fortran standards committee, which contains explicit vector operations. An advantage of PFC over the automatic vectorizing compiler scheme is that, since the translation from Fortran to Fortran 8x is done only once or twice, the translator need not be as efficient as a vectorizing stage embedded in a compiler must be. Instead, the translator can attempt substantially more ambitious program transformations. A second advantage is that if the translation fails to discover a potential vector operation in some critical program region, the programmer can correct the problem directly in the Fortran 8x version rather than trying to recode the input so that the translator will recognize it. This is particularly important since there are some loops which, because of the underlying structure of the problem being solved, can be directly converted to vector form without error, even though the inherently sequential semantics of Fortran makes them difficult or impossible for a translator to convert. Such loops are often easy to recode as explicit vector statements in Fortran 8x. [Allen Kennedy 1982]

PARSE (PARallel Software Environment)

The following information is based on [Casavant Dietz Schwederski Sheu Siegel 1987].

PARSE is being designed to be a software environment for reconfigurable non-shared memory parallel machines. It consists of an integrated collection of language interfaces, as well as debugging and analysis tools. Providing a choice of language interfaces is important because it gives the programmer the opportunity to select the most appropriate or natural specification of a

solution to a problem, thus best utilizing the programmer's time. PARSE is being designed with the hopes of improving the productivity of programmers of parallel systems, where productivity is characterized by three factors: (1) reducing the development time of parallel software, (2) improving the performance and efficiency of parallel software, and (3) improving the reliability of parallel software.

One tool which is always used in program development under PARSE is XPC (explicitly parallel C) - every program is eventually expressed as XPC code. The XPC language is a high-level language being designed to directly specify, in as clean and portable a way as possible, algorithms using (1) explicitly parallel control, (2) data allocated to local memories, and (3) program-controlled machine reconfiguration (subdivisions of groups of processors and SIMD/MIMD mode selection).

PARSE is being developed by the designers of the PASM parallel processing system (Section 3-2). Since one of the goals of PASM is to be a research tool for studying SIMD/MIMD parallelism, and one of the goals of PARSE is to be portable for a class of machines, the XPC language itself will not be PASM-specific, but PASM-specific extensions to XPC will be available to users in much the same way that assembly language programs can be accessed from C code.

PARSE will provide a tool called XPAT (explicitly parallel algorithm analysis tool) that permits analysis of algorithms which are specified in a form which can then be semi-automatically transformed into XPC code. The analysis provided has three uses: (1) to allow the user to debug interprocess communication and synchronization aspects of asynchronous computations without requiring instrumentation of the target hardware environment, (2) to support analysis of efficiency of algorithms to permit the user to make intelligent modifications to improve the use of system resources, and (3) to evaluate the performance of an algorithm itself.

PARSE will also provide software tools to perform automatic parallelization of sequential code.

Pdbx

Pdbx, developed by Sequent Computer Systems, Incorporated, is an enhanced version of dbx [UNIX 1984] that supports debugging of multiple process applications on Sequent's shared

memory multiprocessor machine (Section 3-1). In addition to the functionality of dbx, Pdbx supports the debugging of multiple UNIX processes. Supported are such features as breakpoints for one or more processes, independent examination and tracing of individual processes, and the use of multiple terminals or "windows" for monitoring multiple processes. While Pdbx provides no facilities to control the repeatability of a parallel program, it does extend the functions of a traditional serial symbolic debugger to provide some tools for probing the execution of parallel programs. [Padua Guarna Lawrie 1987] [Sequent 1986] [Sequent 1987]

PISCES

The following information is based on [Pratt 1985] and [Pratt 1987].

PISCES (Parallel Implementation of Scientific Computing EnvironmentS) is an environment for programming parallel machines. It has been designed with the main goal of being able to be implemented reasonably efficiently on a variety of MIMD parallel computers. The primary target for the programming parts of the environment are scientific and engineering applications. The PISCES environments are based on Fortran 77 and UNIX as the underlying sequential language and operating system, respectively.

Dr. Terrence Pratt (designer of PISCES) is currently working on the second version of PISCES, called PISCES 2. PISCES 2 is implemented on the Flexible FLEX/32, a twenty processor machine with both shared and local memory. PISCES 1 was implemented in 1984 on a VAX under UNIX. Parallelism was simulated using UNIX processes. Plans for PISCES 3 are underway. PISCES 3 will be implemented on a hypercube machine and will emphasize parallel I/O and data base access.

SCHEDULE

The following information is based on [Dongarra Sorensen 1987].

SCHEDULE is a package of routines that provide an interface between Fortran programs and a parallel machine. The Fortran routines communicate with shared variables. The programmer defines the dependency relations between the routines (via SCHEDULE calls), and then

SCHEDULE maps the program onto the available hardware in an appropriate way for parallel execution.

SCHEDULE is designed to be a portable environment for developing parallel Fortran programs. Programmers familiar with Fortran can get help from SCHEDULE in implementing a parallel algorithm in a manner that will lend itself to transporting the resulting program across a variety of parallel machines. Existing Fortran subroutines can be called through SCHEDULE, without modification, thus allowing users access to a wide body of existing library software in a parallel setting. Machine intrinsics are invoked within the SCHEDULE package. While a great deal of effort may be required by the designers of SCHEDULE to move it from one machine to another, SCHEDULE users are relieved of the burden of modifying each code he/she wishes to transport from one machine to another.

The designers of SCHEDULE regard it as a temporary solution to one of the primary software problems. SCHEDULE's purpose is to allow the immediate exploitation of existing hardware. They believe that the real hope for a solution lies with new programming languages or perhaps with the 'right' extension to Fortran.

Versions of SCHEDULE are running successfully on the VAX 11/780, Alliant FX/8, and Cray-2 computers. That is, the same user code executes without modification on all three machines. The only modifications made are some minor modifications to SCHEDULE.

SeeCube

SeeCube is a system which monitors and evaluates parallel program execution on an NCUBE hypercube system (Section 3-1). SeeCube is **not** an interactive monitor. It records information about the execution of a parallel program, and then provides a graphical color display of this information in playback mode. [Cybenko 1986]

CHAPTER 9

SOFTWARE LIFECYCLE

Introduction

Many variations of the software lifecycle for *serial* machines can be found in the literature. They all consist of essentially the same phases as follows (based on [Conte Dunsmore Shen 1986]):

(1) **Feasibility of the project.** A decision has to be made as to the realizability of the project.

(2) **Requirements and specifications.** A complete specification of the required functions and performance characteristics of the system should be produced in this phase. Resource needs and preliminary budget estimates should be addressed as well.

(3) **Product design.** The overall system configuration, the implementation language, major modules and their interfaces, data structures, and a testing plan should be specified in this phase.

(4) **Detailed design.** A more detailed module specification should be produced in this phase, including their expected size, the necessary communication among modules, algorithms to be used, and internal data structures. A plan for testing the modules should be specified as well.

(5) **Programming/coding.** An implementation of the modules in the chosen language(s) should

be produced in this phase. Unit testing should be performed.

(6) **System integration.** The integrated modules should be subjected to extensive testing to ensure that all functional requirements are met.

(7) **Installation/acceptance.** The product is delivered to the user organization for final acceptance tests within the operational environment for which it is intended. Documentation and user manuals are delivered and training is conducted.

(8) **Maintenance and Modifications.** Additional discovered errors are corrected, changes in code and manuals are made, new functions are added, and old functions are deleted.

(9) **Obsolescence.** A decision is made that the program is no longer needed or usable.

In addition, each phase of the software lifecycle should be culminated by a verification and validation of activity whose objective is to eliminate as many problems as possible in the products of that phase, and consequently reduce the cost of maintenance, which currently accounts for 70-80% of system costs. Furthermore, one should remember that tests are not proofs of correctness, and they should never be performed by the people that wrote a particular module.

The above software lifecycle can be applied to parallel machines with minor modifications. These modifications take the form of new software development issues that have arisen with the advent of parallel computers.

9-1 Software Development Issues for Parallel Computers

The primary references for the material in this section are [Howe Moxon 1987], [Szymanski Mueller-Wichards 1987] and [Welch 1984], except as noted.

It is intrinsically more difficult to develop software for parallel computers than for serial machines. In addition to the usual challenges, the software engineer must consider communication

overhead and synchronization problems. He/she must be concerned with problem decomposition, the problem of allocating code to the various processors, a task which can be done reliably only *after* the software has been developed, resulting in costly post-development tuning. Memory contention problems are encountered in which different processors attempt to simultaneously access memory. Depending on which processor gets to a memory location first, the outcome of the program may vary, possibly invalidating the output of the program.

Many dialects of conventional sequential languages have been developed, as well as new parallel languages. This has led to a portability problem. In addition, different communication primitives are used on different parallel computers, leading to major software incompatibilities.

The parallel algorithm designer is faced with the problems of having to recognize the realizable parallelism inherent in the problem, having to define the problem data structure for maximum parallel data manipulation, and having to order the sequence of operations for maximum parallel functional operation. Parallel software designers must know both the strengths and weaknesses of their parallel processor to effectively capitalize on its full potential. He/she is most often times burdened with the need to understand the underlying architecture in order to exploit its potential power.

Different levels of parallelism are exhibited in programs. Some have many small inner loops of instructions that interact, while some have large outer loops with completely independent iterations. The software engineer's main objective in dividing up work among processors is to select a level of parallelism that closely matches that of the target computer system. A commonly used measure of parallelism is *granularity*. Recall from Section 1-8 that the granularity of a program indicates how much computing each processor can do independently in relation to the time it must spend exchanging information with other processors. It is often desirable to match the granularity of a program with that of a machine's.

In fine-grained applications, few instructions are executed between communication steps. The ratio of computation to communication is low. On the other hand, coarse-grained applications can be divided up into long independent computing sequences with little interaction between processors. The ratio of computation to communication in coarse-grained applications is high.

There is a big problem with the volume of material that may be produced during program execution. Supercomputer applications use enormous amounts of computation. Consequently, the answers produced in typical supercomputer applications may also be huge ("answer" is defined as the information needed to understand the computed solution, not the total set of numerical results computed). [Rice 1987] illustrates this point with three applications, two real ones from 1983 and 1985 and a hypothetical one from 1995.

1983 Application: The high-speed impact of two steel cubes into a block of aluminum.
This problem is eight-dimensional, with three space variables, time, and four dependent variables (temperature, pressure, the density of steel, and the density of aluminum). The computation used 30 minutes of Cray 1 computer time to cover 2.5 microseconds of real time. This represents roughly 150 billion instructions (12-nanosecond cycle time), including roughly 18 billion arithmetic operations (10 MFLOPS). The answer can be represented by data on a $100 \times 80 \times 80$ special grid for 150 time steps; each of the 96 million grid points has four values (64 bits long). The answer requires only 4.5% of the numbers computed. Effective color plots were used for presenting information about the results. The size of the answer is 3 gigabytes, which is close to the entire disk space on many large-scale systems.

1985 Application: Accretion of material into a black hole (two-dimensional model).
This computation demonstrates the evolution of a black hole over a period of millions of years. Axial symmetry is assumed in order to reduce the problem to a feasible size. The answer consists of 1.25 billion numbers (10 gigabytes). Color movies are discussed as a means of viewing the results. A good-quality movie would require considerably more computation and produce a considerably larger answer than the original computation which provides only moderate resolution in time and space. Modest resolution, slow motion requires 250 Kbytes/sec, while high resolution, normal motion requires roughly 20 Mbytes/sec. [Rice 1987] estimates that a three-dimensional black hole model giving comparable accuracy would have about 1.5 terabytes in the answer, thus producing a 100-hour movie with normal motion and modest resolution.

1995 Application: Tank battle simulation.

This hypothetical application focuses on the weapon systems, the targeting systems, the armor, and the defensive systems of six tanks. Special events such as a shell hit, laser strike, or mine explosion are computationally analyzed in intense detail. The physics of one of these special events is followed at the level of the shell explosion, shell case fragmentation, and attempted penetration of the armor by blast pressure and heat. Other aspects, such as the mechanics of the tanks or terrain, are simulated at a much coarser level. The computation requires one hour of real time or 2 mega-giga instructions (2 nanosecond cycle, 1000 processors) and 700 MFLOPS (200 teraFLOPS machine). The size of the answer is estimated to be about 1 million megawords or 8 terabytes. The answer would be shown, in full, as a color movie with normal motion and high resolution and a duration of about 100-120 hours.

Once the answer has been computed and resides in the supercomputer system, how long will it take to move the answer to the user's location? Peak and effective transfer rates of various facilities are as follows:

Facility	*Peak Rate* (bits/sec)	*Effective Rate* (bits/sec)
Telephone	300	300
2,400-baud line	2,400	2,400
9,600-baud line	9,600	9,600
ARPAnet	57K	20K
Bus on VAX 11/780	1M	160K
Ethernet	10M	1.5M
CDC LCN	50M	3M
Cyber 205 channel	200M	100M

Using the effective transfer rates and the size of the answers, the results look like the following:

Facility	*1983*	*1985*	*1995*
Telephone	3 yr	9 yr	6 millennia
9,600-baud line	1 mo	3 mo	2 centuries
ARPAnet	2 wk	7 wk	1 century
VAX 11/780	2 days	6 days	7 yr
Ethernet	5 hr	15 hr	16 mo
Cyber 205 channel	4 min	13 min	1 wk
Run time	30 min	1 hr	1 hr

Systems separating the user from the supercomputer by two ethernets and a VAX are obviously completely unable to provide reasonable supercomputer service by most people's standards. You will not find too many scientists with the patience to wait a week to see the results of a 30-minute computation. To make matters worse, existing programming environments are so grossly inadequate for today's supercomputers that once the answer is available "locally," the user has neither a place to put it nor adequate means to review it.

9-2 Product Design Phase

The information in this section is mainly based on [Howe Moxon 1987].

Due to the wide variety of modes of parallelism available (e.g., MIMD, SIMD, vector, and dataflow), the various programming methods associated with each mode of parallelism, and the different interconnection networks used (e.g., mesh, hypercube, bus, pyramid, and crossbar), it is necessary to decide on a particular mode of parallelism *before* any detailed design of the system is performed. Three important ideas to base this choice on are: (1) the granularity of an application vs. the granularity of a machine, (2) a shared memory form of communication vs. a message passing form of communication, and (3) the programming languages available.

The granularity of the problem must be well-matched to the architecture in order to exploit the machine's potential power. A system made up of a small number of powerful processors joined by relatively slow communication links is more suitable for coarse-grained applications. A system made up of many simple (in terms of computational power and local memory) processors that communicate relatively fast is more suitable for fine-grained applications. How well the software

approach matches a parallel hardware architecture will affect program development time, the degree of parallelism achieved, and the amount of effort required to maintain the program.

After selecting the proper granularity, the next step is to choose between message passing communications and shared-memory communications, used in distributed memory systems and shared memory systems, respectively. In shared-memory communications, data written by one processor can be read by all other processors in the system. In message passing communications, data is generated by a source processor and delivered to a destination processor. Of the two interprocessor communication methods, message passing is more restrictive since each message must go to a specified recipient. A processor in a shared-memory system need only write a piece of data into a shared location, and then any processor that needs the data can read it.

An important part of choosing a parallel computer is the selection of programming languages available to implement the system in. For example, it may turn out that the computer which seems to be the closest match for the given application in terms of granularity and memory type provides an assembly language as the only language for programming the computer, but the intention was to implement the system in a high-level language. It is important to decide early on in the software lifecycle whether the language(s) offered on possible target architectures are suitable.

9-3 Detailed Design Phase

The information in this section is based mainly on [Miller Stout 1987].

Good software engineering methodologies dictate the use of *abstract data types* (ADTs) for developing cost-effective serial systems. An ADT consists of an abstract data structure (e.g., list, tree, stack, etc.) together with a set of basic operations to be performed on the data structure (e.g., find, insert, push, etc.). The advantage in designing systems in terms of ADTs is that it allows the system to be designed with the essential properties of the data type in mind, but without worrying about implementation constraints and details of the specific machine.

[Miller Stout 1987] introduces a parallel analogue of ADTs, referred to as *abstract data movement operations* (ADMOs). Parallel algorithms can be expressed in terms of fundamental data movement operations without worrying about their implementation or the specific interconnection

of the processors. ADMOs might include operations such as sorting data, routing data, compressing data, and multiple searching.

If a distributed memory machine is chosen as the target architecture, then ADMOs play an important role in the detailed design phase. For fine-grained distributed memory machines (such as the Massively Parallel Processor, a 128 × 128 mesh connected computer discussed in Section 3-1), the physical interconnection topology of the processors will determine the data structure. Thus, efficient operations (i.e., ADMOs) are required to manipulate the data by exploiting the interconnection network. In the event that the problem did not map well to the interconnection topology of the architecture, the software engineer would have to write special data movement operations to simulate other structures. (However, this should not be the case if the architecture was chosen *before* the detailed design phase as proposed earlier.) In medium- and coarse-grained distributed memory machines (such as the Intel iPSC, a hypercube with up to 128 processors discussed in Section 3-1), data structures are important at the "local" level and ADMOs are important at the "global" level. For example, Figure 6-6 illustrates a hypercube algorithm to sum n integers. The program that is stored in each PE uses data structures (e.g., the list of integers to be summed), and the combine phase of the algorithm uses abstract data movement operations (e.g., the routing of data from node 0 to all others, known as a *broadcast* operation, and the routing of data from all nodes to node 0, known as a *report* operation).

If a shared memory machine is chosen as the target architecture, then the software engineer must deal with "tricky" data structure management. For example, memory locations must be locked before they are read from or written to in order to avoid incorrect solutions to problems. If PE i attempts to write to memory location m at the same time PE j attempts to read from memory location m, then depending on which PE gets there first, PE j will have a different value. Shared memory machines are coarse-grained machines by limitations of current technology.

While the idea of ADMOs is to produce one solution on a family of machines, it is important to realize that currently the design phases produce *one* solution on *one* machine; hence, portability problems exist. Automatic vectorizing compilers currently represent the closest programming technique which offers some sort of portability. Programs are written in conventional sequential

languages, and the compiler attempts to extract inherent parallelism wherever possible. This process is not quite as simple as it sounds. The software engineer must modify his/her program so that the compiler will recognize certain structures as being vectorizable, and, of course, each vectorizing compiler is different. ADMOs seem to be the only reasonable way to develop efficient architecture-independent parallel algorithms. However, it has been difficult enough over the years to try and convince programmers of the advantages of using ADTs on serial machines, therefore, we can expect an extremely long acceptance period for ADMOs from the time the concept is first recognized as a possible solution to the portability problem.

9-4 Programming/Coding Phase

The information in this Section is based mainly on [Kuck Davidson Lawrie Sameh 1986] and [Karp 1987].

The software engineer faces new problems in the programming/coding phase of the software lifecycle. First, useful debugging tools are rarely provided for parallel machines. Second, some distributed memory machines do not allow for any I/O from the PEs, making debugging a nightmare. Third, parallel computers are so fast and they produce so much data, that it is impossible to view all of the data in realtime.

Debugging the execution of a parallel computation can be particularly difficult without effective system aids because the logic of the program may be very complex, and the execution may be nondeterministic in that independent operations may be executed in a different order on different runs, exposing bugs in ways that are not always reproducible. In general, programs are more difficult to debug on shared memory systems than distributed memory (message passing) machines because an error usually involves picking up wrong data from a global variable. The processor that picked up the wrong data will continue computing with the bad data, thus producing an erroneous final result. The programmer has no clue as to where the error occurred. Debugging is easier on message passing machines because errors normally cause the system to stop at the point where the error occurred. Regardless of the type of target parallel computer chosen, it is wise to use a simulator to debug programs before getting on the real thing.

9-5 Maintenance Phase

Automatic vectorizing techniques may increase the cost of the maintenance phase of the software lifecycle. Writing code in such a way as to trick the compiler into recognizing structures as being vectorizable goes against all good software engineering techniques and methodologies. Systems may be invalidated the minute a programmer starts shuffling the code around for the automatic vectorizing compiler's sake.

Some of these code systems have been around for a decade or more on three or more generations of hardware, and they have been written in multiple languages. Given the size, complexity, and lifetime involved, it is imperative that all available techniques be utilized to ensure the maintainability of these systems. The software engineer must keep current in such areas as structured design, top-down design, configuration control, "goto-less" programming, and so on. [Rodrigue Giroux Pratt 1984]

CHAPTER 10

SUPERCOMPUTER RESEARCH CENTERS

Introduction

This chapter discusses various supercomputer research centers which develop research parallel-processor systems and/or make commercial supercomputers available to scientists.

THE CONTENTS OF THIS CHAPTER MAY CHANGE WITH THE NEXT EDITION OF THIS DOCUMENT, DUE TO BE RELEASED IN LATE 1988.

10-1 The National Science Foundation's Role in Supercomputing

The following information is based on [Bloch 1987] and [Brandt 1987].

A few years ago, the National Science Foundation (NSF) stated that its most important goal was to educate tomorrow's scientists and engineers about supercomputers. Today (1987), the NSF supports six supercomputer research centers in the United States. In addition, a growing national network has been established that will connect the centers with all of the major universities and the researchers with each other.

The names and locations of the six NSF-supported supercomputer centers are (1) the *San Diego Supercomputer Center* (SDSC) in La Jolla, California, (2) the *National Center for Atmospheric Research* (NCAR) in Boulder, Colorado, (3) the *National Center for Supercomputer Applications* (NCSA) on the campus of the University of Illinois in Champaign-Urbana, (4) the *Pittsburgh Supercomputer Center* (PSC) in Pittsburgh, Pennsylvania, (5) the *Center for Theory and Simulation* on the campus of Cornell University in Ithaca, New York, and (6) the *John von*

Neumann Center (JVNC) near Princeton, New Jersey. Each center is equipped with state-of-the-art supercomputing equipment and a staff to operate and maintain it, as well as to provide user services. Regular in-house training classes are conducted in each center for new users, and intensive two to four week supercomputer workshops are offered each summer for several hundred students.

The NSF announced a new initiative to establish university-based "science and technology centers." The main goal of a center will be to exploit opportunities in science where the complexity of the research problems or the resources needed to solve these problems require the advantages of scale, duration, or facilities that can be provided only by the center mode of research. Funding for the first center is expected to be available in fiscal year 1988.

We will now discuss briefly each one of the NSF-supported supercomputer centers.

The San Diego Supercomputer Center (SDSC)

The following information is based on [Hobson 1987], [Maisel 1987], [Pfeiffer 1985] and [SIAM 1987].

The San Diego Supercomputer Center features a Cray X-MP/48 supercomputer with four processors and an SCS-40 computer (Section 3-1) with one processor. A Cray Y-MP/832 (8 processors, 32 million words of memory) is planned to be installed in 1989. (The Y-MP is the newest line of supercomputers manufactured by Cray Research, Incorporated. The Y-MP is still in the design phases.) The Y-MP will most likely be preceded by an interim machine in 1988. The Center received funding for $100 million over five years, plus an additional $100 million in cost sharing by consortium members, vendors, industrial participants and the State of California.

The scientific staff at SDSC includes doctoral level physicists, chemists, biologists, mathematicians, engineers, and computer scientists. These scientists's job is to evaluate, convert, install, optimize and develop software for users in their respective disciplines, with the goal of making the broadest and best possible use of the computing resources at SDSC. Users of SDSC carry out research in biology and chemistry, mathematics and computer science, engineering, geophysics, and physics.

The SDSC has been conducting an educational effort to bring new users up to speed as part of the mission to make supercomputing available to academic researchers. Two-day introductory workshops are given each month, and a two-week summer institute was held in August 1987. Approximately 4000 people are on the mailing list for the center's newsletter titled *Gather/Scatter*.

More information may be obtained from Wayne Pfeiffer, San Diego Supercomputer Center, GA Technologies Inc., P.O. Box 85608, San Diego, CA 92138.

The National Center for Atmospheric Research (NCAR)

The National Center for Atmospheric Research did not respond to information requests on the Center, and there is no literature currently available on the Center in any journals or conference proceedings.

The National Center for Supercomputing Applications (NCSA)

The following information is based on [NCSA 1987], [SIAM 1987] and [Wilhelmson 1985].

The National Center for Supercomputing Applications was established in February 1985 with a five-year grant from the National Science Foundation. It serves the national research community by offering a comprehensive computing system and opportunities for training and interaction within an interdisciplinary research environment. NCSA features a Cray X-MP/48 supercomputer with four processors and 8 million words of memory.

The *Interdisciplinary Research Center* (IRC) of NCSA, located on the campus of the University of Illinois, allows supercomputer users to come together to learn, experiment, and exchange ideas about new research and new computational techniques. Scientists, engineers, and scholars from many disciplines interact with the NCSA staff of research scientists, computer professionals, graphic specialists, and user consultants. The IRC offers state-of-the-art, networked workstations for interactive work on the Cray X-MP and for other research purposes such as analyzing results, processing graphical images, and preparing reports and publications.

Although consultants are available to provide guidance in optimizing code and to direct researchers to vectorized algorithms, it is the responsibility of every user to optimize programs for

execution on the Cray X-MP. The consulting staff provides short courses for users of the NCSA facilities on such topics as the basic system, vectorization, debugging, mathematical software, and using the Cray in a workstation environment.

An *Industrial Supercomputing Program* has been established within the NCSA. This program is designed to expand the benefits of NCSA's interdisciplinary research effort to include researchers from a select group of corporations. Designated employees from participating corporations receive training tailored to their needs, access to state-of-the-art workstations and appropriate software, and technical staff consulting and support. Corporations make a three year commitment to the Program at an annual fee of $1 million.

A *Scientific Visualization Program* has been designed to provide scientists with the ability to produce videotapes and films of research results, to playback 50-second segments of a simulation in real-time, and to interact and explore data. This program is based on an Alliant FX/8 minisupercomputer with eight processors, 128 megabytes of memory, and a Fortran compiler which automatically vectorizes code and can run programs across multiprocessors down to the loop level, thus permitting the faster processing of data. Attached to the FX/8 are two Raster techology frame buffers that are used for the display of color images at two times the normal television resolution. Connected to these frame buffers on a video pipeline is an Abekas A62, a digital video storage device used to play back images exceeding real time.

One of the major goals of the NCSA is to take the best of today's technology and integrate it to support supercomputer users. Towards this goal, the NCSA has successfully developed an integrated workstation environment on the Apple Macintosh Plus microcomputer for the Cray X-MP. Workstation functionality includes good terminal emulation and communication with the Cray X-MP, manipulation of information as text or graphics, and easy transfer of research data between the Cray X-MP and application programs on the Macintosh.

More information may be obtained from the NCSA Visitors Program, University of Illinois, 152 Computing Applications Building, 605 East Springfield Avenue, Champaign, IL 61820 or call (217) 244-0074.

The Pittsburgh Supercomputing Center

The Pittsburgh Supercomputing Center's mission is to provide state-of-the-art computing resources to the national scientific and engineering research communities in a reliable and convenient way, to educate researchers about the benefits of supercomputing, and to introduce industrial users to the benefits of supercomputing. The Center features a Cray X-MP/48 supercomputer with four processors intended particularly for engineering and scientific research, and an Alliant FX/8. [SIAM 1987] More information on the Center may be obtained from Dr. Georgette H. Demes, (202) 357-9776.

The Center for Theory and Simulation in Science and Engineering

The Cornell Theory Center (CTC) was established to provide supercomputing resources for researchers nationwide. The center provides computing resources consisting of an IBM 3090-600E (a powerful four-processor mainframe) with six vector facilities and five attached scientific computers from Floating Point Systems, several workstations, drafting plotters and printers. New and experienced remote users of the CTC can receive training at the Cornell campus on a quarterly basis. Local users can attend training workshops throughout the spring and fall semesters. Workshops provide lectures and demonstrations for CTC system features, and hands-on sessions with consulting and pre-planned exercises ensure that students gain real experience. Workshops are also given on such special topics as vectorization and parallelization. [Brown Siegel 1985] [Cornell 1987] [SIAM 1987]

More information may be obtained from the Executive Director, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY 14853.

The John von Neumann Center for Scientific Computing

The John von Neumann Center for Scientific Computing is located near Princeton. It is being developed to provide state-of-the-art computing and communications to university, government, and industrial researchers. The center features two Cyber 205s and an ETA^{10} supercomputer. [Orszag 1985] [SIAM 1987]

More information may be obtained from Brendan McNamara, Executive Vice President, John von Neumann Center, P.O. Box 3717, Princeton, NJ 08540.

10-2 Supercomputing at Florida State University

The information in this section is based on [Lannutti 1985] and [SCRI 1986].

The Supercomputer Computations Research Institute (SCRI) is located on the Florida State University campus in Tallahassee, Florida. The SCRI's goal is to provide scientists in a variety of fields with the means of exploiting supercomputers as effective tools in their research. It features a CDC Cyber 205 with 2 vector pipelines and an ETA^{10} supercomputer with four processors. The SCRI is funded by Federal, State, University and industry agencies.

Research, consultant and support staff are available by phone, electronic mail and personal visit. This staff can provide assistance in code conversion and optimization, algorithm selection, and specific content area applications. A sequence of workshops is offered for new users and are offered several times a year. A series of weekly seminars is presented on advanced topics in supercomputer applications. Conferences, summer schools and workshops in a variety of related areas are conducted periodically.

10-3 Supercomputing at the University of Illinois at Urbana-Champaign

We already discussed the NSF-supported NCSA in Section 10-1, which is located on the University of Illinois campus. The Center for Supercomputing Research and Development (CSRD), established in late 1984, is also located on the University of Illinois at Urbana-Champaign campus. Researchers at the CSRD are building the Cedar system, a prototype shared-memory multiprocessor that consists of clusters of eight processors. The Cedar system will be used to explore parallel processing performance, develop software to exploit shared-memory parallel processors effectively, and provide high performance over a wide range of applications. Software projects include the development of a UNIX-based multiprocessor operating system called Xylem, the development of the Parafrase restructuring compiler which will be able to transform serial as well as parallel Fortran constructs to exploit the system, dealing with both vector and

multiprocessing parallelism, and the development of parallel algorithms for engineering and scientific applications. In addition, a supercomputer programming environment called Faust is being developed and consists of an editor, interactive optimizing compiler, parallel debugger and performance analyzer/monitor, each of which uses a common graphics-based user interface. [CSRD 1986] [Lelick 1985]

10-4 Supercomputing at the Supercomputing Research Center (SRC)

The Supercomputing Research Center (SRC), established in November 1984, is located in Lanham, Maryland. The SRC is funded by the National Security Agency (NSA). It carries out research and development on national security programs as well as unclassified research in parallel processing algorithms and systems. Conferences and workshops are given yearly. [Schneck 1985]

10-5 Supercomputing at the Advanced Computing Research Facility (ACRF)

The following information is based on [IJSA 1987].

The Advanced Computing Research Facility (ACRF) is an Argonne National Laboratory which currently houses seven multiprocessors available to scientists to experiment with innovative machines and to develop software tools for state-of-the-art computers. The ACRF was established in 1984, and it was initially equipped with two multiprocessors: the Denelcor Heterogeneous Processor (HEP) and a locally built system with eight processing units. Since then, the ACRF has installed five new commercial multiprocessors, four of which are mentioned in [IJSA 1987]: (1) an Alliant FX/8 system with 8 vector processors sharing 32 Mbytes of memory, (2) an Encore Multimax system with 20 processors sharing 20 Mbytes of memory, (3) a Sequent Balance 21000 system with 24 processors sharing 16 Mbytes of memory, and (4) an Intel iPSC four-dimensional hypercube system.

These five machines will be used to address a variety of questions in parallel processing, including (1) What architectures are best suited for a specific application? (2) How can computations be organized to exploit the full potential of a machine? and (3) Can we develop transportable algorithms without sacrificing performance? Argonne scientists are conducting

research in advanced computing that simultaneously emphasizes new algorithms, new architectures, and improved computing environments in order to answer these questions.

Research on these multiprocessors is not limited to Argonne scientists. An extensive visitors' program is provided, and classes on advanced computing are frequently given. A summer institute is being planned for graduate and postdoctoral students, as well as a workshop on performance evaluation of parallel computers and programs. Scientists at other research institutes are encouraged to use the computers.

10-6 Supercomputing at the University of Calgary in Western Canada

The following information is based on [Nunns 1987].

A Control Data Cyber 205 supercomputer was installed on the University of Calgary campus (Alberta, Canada) in late January 1985. It now serves academic researchers, commercial clients, and nonprofit users across Canada. Funding for the supercomputer project at the University of Calgary included $12 million for the Cyber 205 and $5 million to be used over five years for software, maintenance, and user services. Grants of supercomputer time are allocated to any person or corporation, for potentially commercially viable research and development purposes.

The department of SuperComputing Services (SCS) was established at the University of Calgary in order to provide all necessary user services. The SCS staff consists of highly trained people in the areas of engineering, geography, geology, mathematics, meteorology, physics, and computer science. These scientists are available for user consultation, running benchmarks, or assisting with the analysis of applications programs. Experienced applications analysts give user courses in usage of the Cyber 205, as well as vector programming in Fortran and C.

A variety of manuals and other forms of documentation are produced and distributed by the SCS. In addition, the SCS publishes a quarterly publication titled *Super*C Newsletter*. This publication introduces new hardware and software options, outlines applicable conferences and courses, and details projects in which users are involved.

10-7 Supercomputing at the California Institute of Technology (Caltech)

The California Institute of Technology (Caltech) does a great deal of research in parallel processing. Dr. Geoffrey C. Fox is well known as the designer of the Cosmic Cube, the first working hypercube. Six scientists from Caltech recently published the book *Solving Problems on Concurrent Processors*, which describes the concurrent implementation of a number of algorithms widely used in scientific computing. A package called *The Software Supplement* has been developed and consists of (1) a hypercube simulator which runs under the UNIX, ULTRIX, VMS, and PC XENIX operating systems, (2) a set of 20 application programs in both C and Fortran which run on the simulator and illustrate the concurrent algorithms described in *Solving Problems on Concurrent Processors*, and (3) a book containing a description of the simulator, source code for each of the application programs, and explanatory notes. A videotape course based on the book "*Solving Problems on Concurrent Processors*" has been developed. [Caltech 1987]

10-8 Supercomputing at the University of Virginia

The Institute for Parallel Computing, located at the University of Virginia, Charlottesville, was established in 1987. One of the two primary issues it will focus on is programming environments for parallel architectures. The PISCES II programming environment (Section 8-2), developed by Dr. Terrence Pratt, will be ported to a 64-node hypercube in the near future. The primary funding for the institute is provided by the Joint Tactical Fusion Office and the Jet Propulsion Laboratory. [SIAM 1987]

10-9 Super Advantage

Super Advantage is a program set up by the Houston Area Research Center (HARC) in Woodlands, Texas which allows people to access an SX-2-400 (Section 3-1) supercomputer "for a fraction of what you'd expect to pay for normal time-sharing." Members of Super Advantage are entitled to (1) 10 hours SX2 CPU time per month, (2) 10 man/days training in HARC short courses, (3) unlimited extended memory use, (4) 1 membership on the policy board, (5) 500 Mbytes permanent storage, (6) MicroVAX II configuration with Ultrix and sofware development

tools, and (7) three days on-site training. More information may be obtained from the Computer Systems Applications and Research Center, Houston Area Research Center, 10077 Grogans Mill Road, Suite 550, Woodlands, Texas 77380 or call (713) 363-7981. [Supercomputing 1987a]

CHAPTER 11

RELEVANT RESEARCH TOPICS

Introduction

During the Summers of 1985 and 1986, the Supercomputing Research Center (SRC) (Section 10-4) held two workshops, one on supercomputing and one on parallel algorithms and architectures, respectively. Top scientists from around the world were invited to these workshops for the sole purpose of discussing the state of parallel computing from both a hardware and a software point of view. The final products were two reports written by the invited research scientists on what areas should be researched by the SRC over the five years to come.

The information in this chapter is based on [SRCa 1986] and [SRCb 1986]. (The Summer 1985 workshop's report was not completed until February of 1986, and the Summer 1986 workshop's report was written up by December of 1986, hence the references are both dated 1986.)

11-1 Standardized Classification Structure

A standardized classification structure should be developed for parallel algorithms. This structure will allow researchers to predict what class of algorithms will best match a given architecture. A proper mapping between the architecture classes and algorithm classes will guide the direction of future supercomputer design and selection.

One particular project which requires a better understanding of which sets of programs execute efficiently on which class of architectures is an *architecture compiler*. Such a software tool would allow specification of an application algorithm at a high level and would produce a detailed

description at a much lower level of an architecture on which that algorithm would execute efficiently.

Generalized software monitors directed toward understanding performance across systems are also important. These will help to understand why classes of applications perform more efficiently on some architectures than on others.

11-2 Public Library of Developed Algorithms

A public library of developed algorithms should be constructed. This library will allow researchers to avoid repeating work, to collect and disseminate state-of-the-art knowledge, and to help teach algorithms. Specific features of such a library include the following:

(1) The library should contain fundamental algorithms of general use, represented in a single standard notation (if possible) that may well contain a pictoral component. This notation will be important in its own right, since it represents a step toward standardization, and can be used by automatic tools for program transformation.

(2) Each algorithm in the library should be classified, cross-referenced to related algorithms, and include a history of its implementations and their behavior.

(3) Access to the library should be public, computer-based, and supported by a professional team of librarians.

Recall that Cray Research, Inc. publishes the Cray Directory of Supercomputer Applications Software (Section 8-2). It is not known at this time if this directory adheres to the specifications given above.

11-3 Theory of Complexity

A theory of complexity based on realistic models of computation should be developed. This theory

will allow researchers to avoid searching for impossible algorithms and to discover the limits of parallelism. This theory should give researchers an understanding of the tradeoffs among memory, computation, communications, I/O, and the accuracy of results.

11-4 Programming Environments

Because of the inordinate amount of time otherwise needed to understand and effectively use parallel computers, good, flexible, integrated programming environments are needed. Such environments will assist in and automate the following tasks of problem solving: application/problem definition → algorithm → high-level code → execution → analysis of results. Research in this area should be directed toward creating, sustaining, and improving upon an effective problem-solving environment which will transfer the burden of auxiliary tasks and bookkeeping to our software tools.

Note that quite a few parallel programming environments are already under development, including Faust, PARSE, PISCES, and SCHEDULE (all discussed in Section 8-2).

11-5 Programming Languages

New programming languages that express all levels of parallelism and concurrency and their translation to selected target architectures are needed. Languages for representing parallel algorithms should be researched from a number of perspectives. Ideally, a language should be suited for programming and expressing parallel constructs. Moreover, notational constructs should be developed to express specific notions of computation as implemented and executed on target families of parallel machines. The expression of parallelism should not introduce superfluous barriers and critical sections, as might be encountered in a naive parallelization of standard serial code. Constructs should be developed to represent probability and indeterminacy along with the more standard flow controls to model real processes and capture actual performance. Abstract data types and their equivalents (abstract data movement operations (ADMOs) as discussed in Section 9-3), as they evolve in parallel settings, ought to be readily accessible in such a high level representation.

11-6 A Vocabulary and Notation for Parallelism

An insufficient vocabulary currently exists with which to describe parallelism, either in algorithms or architectures. What cannot be described clearly cannot be understood clearly. A vocabulary and notation for parallelism which is robust and complete and shows promise of being extensible to include new machines and new technologies is needed.

11-7 Software Support for Parallel Programming

It is generally acknowledged that the debugging of parallel programs is significantly more difficult than debugging sequential programs. Software debugging tools need to be developed to help the programmer. Both compile-time and run-time tools are needed. Compile-time offers the best opportunity to detect as many errors as possible. Compile-time error detection is very difficult in languages like Fortran with aliasing and side effects. In addition, the quantity of information produced is enormous and hard to manage. Sufficient declarations added to the code by the user may make this whole problem more manageable. A compile-time debugger should warn the programmer about actual or potential errors. Interactive use of a compile-time debugger may enable the programmer to skip much of the analysis information and look only at the relevant part. A graphical interface will be essential.

A simulator of the parallel system capable of providing reproducible results is needed. Some kinds of run-time errors can be uncovered using simulations. For debugging on the actual parallel system it is not clear what features will be necessary beyond extensions to sequential debuggers for both small and large number of processors. In shared memory systems where updates to shared variables are not sequential (i.e., race conditions exist), a tracing mechanism that determines the order in which tasks update a shared variable will be critically needed to detect these potential problems.

CHAPTER 12

JOURNALS AND BOOKS

Introduction

This chapter gives lists of journals and books related to parallel processing. These lists are by no means complete, but they may guide the reader to further information.

THE CONTENTS OF THIS CHAPTER MAY CHANGE WITH THE NEXT EDITION OF THIS DOCUMENT, DUE TO BE RELEASED IN LATE 1988.

12-1 Journals

The following journals are either completely dedicated to the field of parallel processing/supercomputing or periodically provide related articles. Wherever possible, descriptions of each journal are given along with subscription fees and an address to write to.

Complex Systems

This journal brings together a broad range of research on all aspects of the theory and applications of complex systems. It publishes summaries of completed research projects, full-length articles, and notes reporting specific results. Individual subscriptions are $65 for six issues. Subscription orders should be sent to Complex Systems Publications, Inc., P.O. Box 6149, Champaign, IL 61821-8149.

IEEE Computer

This journal is published by the Computer Society of the Institute of Electrical and Electronics Engineers (IEEE). An annual subscription is included in society member dues.

Computer Architecture News

This journal is published by the Association of Computing Machinery Special Interest Group on Computer Architectures (ACM SIGARCH). Subscriptions costs $20 a year for members of the ACM and $34 for non-members.

The Computer Journal

This journal is published by The British Computer Society, Cambridge University Press. Subscriptions cost $275 for six issues.

Computer Languages

This journal is published by Pergamon Journals Limited. Subscriptions cost $157 for individuals and $615 for institutions.

Computing Surveys

This journal is published by the Association of Computing Machinery (ACM), New York.

IEEE Software

This journal is published by the Institute of Electrical and Electronics Engineers (IEEE). Subscriptions cost $16 in addition to any IEEE group or society dues and $25 for members of other technical organizations.

IEEE Transactions on Computers

This journal is published by the Institute of Electrical and Electronics Engineers (IEEE). Write to IEEE Headquarters, 345 East 37th Street, New York, NY 10017 for more information.

International Journal of Parallel Programming

This journal addresses programming challenges by parallel computing systems, including linguistic foundations, implementation techniques, software engineering aspects, and performance studies. The cost for individual subscribers is $60 for six issues. Subscription orders should be sent to Plenum Publishing Corporation, 233 Spring Street, New York, NY 10013.

The International Journal of Supercomputer Applications

This journal contains articles on supercomputer applications software for a wide variety of areas, including aerospace engineering, astrophysics, cryptographic analysis, pharmacuetical research, and molecular biology. The journal is published quarterly and costs $50 per individual. Subscription orders should be sent to MIT Press Journals, 55 Hayward St., Cambridge, MA 02142.

Journal of Parallel and Distributed Computing

This journal contains original research papers, as well as critical reviews on the design, evaluation, and practices of advanced computing systems. It is directed to researchers, engineers, educators, managers, programmers, and users of computers who have particular interest in parallel processing and distributed computing. The current subscription fee is $119 for six issues. Subscription orders should be sent to Academic Press, Inc., Journal Promotion Dept., 1250 Sixth Avenue, San Diego, CA 92101.

The Journal of Supercomputing

This journal contains articles on technology, architecture and systems, algorithms, languages, performance methods, and applications. The journal costs $49.50 for individuals ($125 for institutions) and is published quarterly. Write to Kluwer Academic Publishers, P.O. Box 358, Accord Station, Hingham, MA 02018-9990.

New Generation Computing

This journal is an international journal on fifth generation computers. It is published by Ohmsha, Limited in Japan. A one year subscription costs $264.

Parallel Computing

This journal is an international journal containing articles on the theory and use of parallel computer systems. It features original research work, tutorial and review articles, and accounts of practical experience with parallel computers. Subscriptions for 1987 were $225.75 for six issues. Subscription orders should be sent to Elsevier Science Publishers Co. Inc., Journal Information Center, 52 Vanderbilt Avenue, New York, NY 10017.

SIAM Journal on Computing

Subscription fees for nonmembers is $116. Subscription fees for SIAM members are substantially less, however. Write to SIAM, 1400 Architects Building, 117 South 17th Street, Philadelphia, PA 19103-5052 for information about becoming a member of SIAM.

SIAM Journal on Scientific and Statistical Computing

This journal contains articles on numerical, statistical, and non-numerical techniques for solving scientific and statistical problems on computers. Emphasis is on the implementation of such techniques with computer languages, adaptive approaches, interactive graphics, data management facilities, new computer architectures, and special-purpose hardware. The journal is published bimonthly and costs $102. Subscription orders should be sent to Customer Service, SIAM, 1400 Architects Building, 117 South

Supercomputer

This journal focuses on supercomputing, especially in the Netherlands, with emphasis on the practical aspects of supercomputing Subscriptions cost roughly $75 and can be obtained from Supercomputer, P.O. Box 4613, 1009 AP Amsterdam, the Netherlands.

SuperComputing

This magazine is designed for both those who use supercomputers as problem-solving tools and those who are exploring the potential of supercomputers. It includes discussions of supercomputer evaluation and articles about specific industry advances in technology. Subscriptions are free and can be obtained from SuperComputing, 510 S. Mathilda Avenue, Suite 4419, Sunnyvale, CA 94086.

12-2 Books

The following is a list of books related to parallel processing. Obviously, there are hundreds of books on the subject, but we only list a few here.

Algorithm-Structured Computer Arrays and Networks: Architectures and Processes for images, percepts, models, information

Leonard Uhr. Academic Press, 1984.

Computer Architecture and Parallel Processing

Kai Hwang and Fayé A. Briggs. McGraw-Hill, 1984.

Designing Efficient Algorithms for Parallel Computers

Michael J. Quinn. McGraw-Hill, 1987.

Parallel Algorithms for Regular Architectures

Russ Miller and Quentin F. Stout. The MIT Press, Cambridge, MA., 1988.

Solving Problems on Concurrent Processors

G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. Prentice-Hall, 1987(8).

BIBLIOGRAPHY

Abraham, J.A., Banerjee, P., Chen, C., Fuchs, W.K., Kuo, S. and Reddy, A.L.N. 1987. Fault Tolerance Techniques for Systolic Arrays. *IEEE Computer* 20, 7 (July), pp. 65-75.

Ackerman, W.B. 1979. Data flow languages. *Proceedings of the National Computer Conference* 48, pp. 1087-1095.

Ackerman, W.B. and Dennis, J.B. 1979. VAL - A Value-Oriented Algorithmic Language. *Preliminary Reference Manual, Laboratory for Computer Science Technical Report 218*, MIT, Cambridge, Massachussetts, (June).

Adams III, G.B., Agrawal, D.P. and Siegel, H.J. 1987. A Survey and Comparison of Fault-tolerant Interconnection Networks for Supersystems. *IEEE Transactions on Computers* C-31, (May), pp. 443-454.

Adams III, G.B. 1987. Fault-Tolerant Multistage Interconnection Networks. *IEEE Computer* 20, 6 (June), pp. 14-27.

Allan, S.J. and Oldehoeft, A.E. 1979. A Flow Analysis Procedure for the Translation of High Level Languages to a Data Flow Language. *Proceedings of the 1979 International Conference on Parallel Processing*, (August), pp. 26-34.

Allen, J.R. and Kennedy, K. 1982. PFC: A Program to Convert Fortran to Parallel Form. *The Proceedings of the IBM Conference on Parallel Computers and Scientific Computations*, pp. 186-203.

Allen, J.R. and Kennedy, K. 1985. A Parallel Programming Environment. *IEEE Software*, (July), pp. 21-9.

Ametek 1987. Concurrent Processing: Hypercube. Ametek Computer Research Division.

Anderson, R. Grimes, R., Riebman, R. and Simon, H. 1987. Early Experience with SCS-40. *Supercomputer* 22, (November), pp. 26-36.

Annaratone, M., Arnould, E., Cohn, R., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O., Sarocky, K., Senko, J. and Webb, J. 1987. Architecture of Warp. *Proceedings of COMPCON Spring 87*, San Francisco, (February).

Annaratone, M., Arnould, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O. and Webb, J. 1987. The Warp Computer: Architecture, Implementation and Performance. *IEEE Transactions on Computers* C-36, 12 (December).

Annaratone, M., Bitz, F., Clune, E., Kung, H.T., Maulik, P., Ribas, H., Tseng, P. and Webb, J. 1987. Applications and Algorithm Partitioning on Warp. *Proceedings of COMPCON Spring 87*, San Francisco, (February).

Arvind, Gostelow, K.P. and Plouffe, W.E. 1978. An Asynchronous Programming Language and Computing Machine. *Department of Information and Computer Science Report TR 114a*, University of California, Irvine, (December).

Arvind and Thomas, R.H. 1980. I-Structures: An Efficient Data Type for Functional Languages. *Laboratory for Computer Science Technical Memo* 178, MIT, Cambridge, Massachussetts, (September).

Baiardi, F., De Francesco, N. and Vaglini, G. 1986. Development of a Debugger for a Concurrent Language. *IEEE Transactions on Software Engineering* SE-12, 4 (April), pp. 547-553.

Baillie, C.F. 1986. A General Fortran to C Translator. *Computing Physics Communications* (Netherlands) 41, 2-3 (August), pp. 409-414.

Baldwin, D. and Quiroz, C. 1987. Parallel Programming and the Consul Language. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 389-392.

Barszoz, E. and Howard, L.S. 1987. Static Data Flow Simulation Study at Ames Research Center. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 427-430.

Batcher, K.E. 1984. Design of a Massively Parallel Processor. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 104-108.

BBN 1986. *Butterfly Parallel Processor Overview* Version 1, Bolt, Beranek and Newman, Inc., (March 6).

Beetem, J., Denneau, M. and Weingarten, D. 1985. The GF11 Supercomputer. *Proceedings of the Symposium on Computer Architectures.*

Bergmark, D. 1987. Programming the FPS T Series. *Cornell Theory Center Technical Report,* (June).

Bhuyan, L.N. 1987. Interconnection Networks for Parallel and Distributed Processing. *IEEE Computer* 20, 6 (June), pp. 9-12.

Bloch, E. 1987. Supercomputing and the Growth of Computational Science in the National Science Foundation. *The International Journal of Supercomputer Applications* 1, 1 (Spring), pp. 5-8.

Bossavit, A. and Meyer, B. 1981. The Design of Vector Programs. *Algorithmic Languages, Proceedings of the International Symposium*, pp. 99-114.

Brandes, T. and Sommer, M. 1987. A Knowledge-Based Parallelization Tool in a Programming Environment. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 446-448.

Brandt, L.E. 1987. The NSF National Computing Centers: Past, Present and Future. *Computer Physics Communications* 45, pp. 147-148.

Brown, A. and Siegel, P. 1985. Cornell NSF Supercomputer Center, Cornell University. *IEEE Software* 2, 6 (November), page 64.

Bruegge, B., Chang, C., Gross, T., Lam, M., Lieu, P., Noaman, A. and Yam, D. 1987. Programming Warp. *Proceedings of COMPCON Spring 87*, San Francisco, (February).

Bucher, I.Y. and Simmons, M.L. 1987. A Close Look at Vector Performance of Register-to-Register Vector Computers and a New Model. *Performance Evaluation Review* 15, 1 (May), pp. 39-45.

Buzbee, B.L. 1986. A Strategy for Vectorization. *Parallel Computing* 3, pp. 187-192.

Caltech 1987. *Concurrent Supercomputing Initiative at Caltech: information package*. California Institute of Technology.

Carpenter, G.F., Tyrrell, A.M. and Holding, D.J. 1986. Guidelines for the Synthesis of Software for Distributed Processors. *Proceedings of the Programmable Electronic Systems Safety Symposium* PE S 3, pp. 164-75.

Casavant, T.C., Dietz, H. G., Schwederski, T., Shen, C-Y. and Siegel, H.J. 1987. Software Plans for PASM. *Proceedings of the Second International Conference on Supercomputing*, (May), pp. 428-439.

Charlesworth, A.E. and Gustafson, J.L. 1986. Introducing Replicated VLSI to Supercomputing: the FPS-164/MAX Scientific Computer. *IEEE Computer* 19, 3 (March), pp. 10-23.

Chen, M.C. 1986. A Design Methodology for Synthesizing Parallel Algorithms and Architectures. *Journal of Parallel and Distributed Computing*, pp. 461-491.

Chin, C. and Hwang K. 1984. Packet Switching Networks for Multiprocessors and Dataflow Computers. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 329-347.

Chu, H., Delp E.J. and Siegel, H.J. 1987. Image Understanding on PASM: A User's Perspective. *Proceedings of the Second International Conference on Supercomputing*, (May), pp. 440-449.

Cline, C.L. and Siegel, H.J. 1985. Augmenting Ada for SIMD Parallel Processing. *IEEE Transactions on Software Engineering* SE-11, 9 (September), pp. 970-77.

CMU 1987. Collection of Papers on Warp. Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, (January).

Coleman, D. 1986. The Evolving Supercomputer. *Hardcopy*, (August), pp. 28-38.

Collado, M., Morales, R. and Moreno, J.J. 1987. A Modula-2 Implementation of CSP. *SIGPLAN Notices* 22, 6 (June), pp. 25-38.

Conner, M.S. 1987. Transputer-based PC add-in board supports programming in C. *EDN*, (August 6), pp. 73-74.

Conte, D., Hifdi, N. and Syre, J.C. 1980. The Data Driven LAU Multiprocessor System: Results and Perspectives. *Proc. IFIP Congress*.

Conte, S.D., Dunsmore, H.E. and Shen, V.Y. 1986. *Software Engineering Metrics and Models*. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA.

Cornell 1987. *The Cornell Theory Center* and *Cornell Theory Center*, Cornell University, Ithaca, New York.

Cray 1987. *The Cray-2 Series of Computer Systems*; *The Cray X-MP Series of Computer Systems* (July) and Cray Research, Inc. *NEWS* (February), Cray Research, Incorporated.

CSRD 1986. *Center for Supercomputing Research & Development Research Review.* University of Illinois at Urbana-Champaign.

CSRD 1987. Parafrase Software Package Available. Parafrase II. Center for Supercomputing Research and Development Bulletin 1, 2 (December), page 3.

Cybenko, G. 1986. *Visualizing Hypercube Computations.* Colloquium announcement. Department of Computer Science, University at Buffalo, State University of New York, (December 4).

Darema-Rogers, F., Pfister, G.F. and So, K. 1987. Memory Access Patterns of Parallel Scientific Programs. *Proceedings of the 1987 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* 15, 1 (May), pp. 46-58.

Datapro 1986. Vector Processing - Advanced Technologies. Datapro Research Corporation.

Datapro 1987. The EDP Buyer's Bible 15, 10 (October).

Davidson, E., Kuck, D., Lawrie, D. and Sameh, A. 1986. Supercomputing Tradeoffs and the Cedar System. *Center for Supercomputing Research and Development Technical Report*, University of Illinois at Urbana-Champaign, (May).

Denning, P.J. 1986. Parallel Computing and its Evolution. *Communications of the ACM* 29, 12 (December), pp. 1163-1167.

Dennis, J.B. 1979. The Varieties of Data Flow Computers. *Proceedings of the First International Conference on Distributed Computing Systems*, (October), pp. 430-439.

Dennis, J.B. 1984. Data Flow Supercomputers. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 480-488.

Deshpande, S.R., Jenevein, R.M. and Lipovski, G.J. 1985. TRAC: An experience with a novel architecture prototype. *National Computer Conference*, pp. 247-258.

Dietz, H. and Klappholz, D. 1985. Refined C: A Sequential Language for Parallel Programming. *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 442-449.

Dimopoulos, N.J., Li, K.F., Wong, E.C., Dantu, R.V. and Atwood, J.W. 1987. The Homogeneous Multiprocessor System. An Overview. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 592-599.

Dongarra, J.J. 1986. How do 'minisupers' stack up? *IEEE Computer* 19, 3 (March), pp. 93, 100.

Dongarra, J.J. and Sorenson, D.C. 1987. A Portable Environment for Developing Parallel Fortran Programs. *Parallel Computing* 5, pp. 175-186.

Drake, B.L., Luk, F.T., Speiser, J.M. and Symanski, J.J. 1987. SLAPP: A Systolic Linear Algebra Parallel Processor. *Computer* 20, 7 (July), pp. 45-49.

Eichholz, S. 1987. Parallel Programming with ParMod. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 377-380.

Elizabeth, M., Hull, C., and Donnan, G. 1986. Contextually Communicating Sequential Processes - a Software Engineering Environment. *Software - Practice and Experience* 16, 9 (September), pp. 845-864.

Elizabeth, M. and Hull, C. 1987. Occam - A Programming Language for Multiprocessor Systems. *Computer Language* 12, 1, pp. 23-37.

Elliott, C.J. 1986. Mapping Sequential Fortran into Parallel Occam. *IEE Colloquium on 'Software Engineering for VLSI Parallel Processors'*, Digest No. 102 (October), pp. 9/1-2.

Emmen, Ad. 1987. ETA10-P: A "Poor Man's" Supercomputer for 1 Million Dollars. *Supercomputer* 22, (November), pp. 4-6.

Emrath, P. 1985. Xylem: An Operating System for the Cedar Multiprocessor. *IEEE Software* 2, 4 (July), pp. 30-37.

Encore. 1987. *Multimax Multiprocessor Systems; The Power of Parallel Thinking*. Encore Computer Corporation, Marlboro, MA.

ETA 1987. *I/O* (An ETA Systems Publication) 4, 1 and 2.

Evans, D.J. 1986. Algorithm Development for Parallel Processing. *IEE Colkoquium on 'Software Engineering for VLSI Parallel Processing'*, Digest No. 1986/102, pp. 4/1-4/7.

Fatoohi, R.A. and Grosch, C.E. 1987. Implementation of Four Color Cell Relaxation Scheme on the MPP, Flex/32, and Cray/2. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 424-426.

Feng, T. 1981. A Survey of Interconnection Networks. *IEEE Computer*, (December), pp. 12-27.

Fernbach, S. 1984. Applications of Supercomputers in the U.S. - Today and Tomorrow. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 421-428.

Flynn, M.J. 1966. Very high-speed computing systems. *Proceedings of the IEEE* 54, 12 (December), pp. 1901-1909.

Forefronts 1988. Parallel Processing on the IBM 3090: Measurements of PFP. *Forefronts* 3, 9 (January), Center for Theory and Simulation in Science and Engineering, Cornell University, pp. 2 -4.

Fortes, J.A.B. and Wah, B.W. 1987a. Systolic Arrays - from Concept to Implementation. *IEEE Computer* 20, 7 (July), pp. 12-17.

Fortes, J.A.B. and Wah, B.W. 1987b. Systolic Arrays: A Survey of Seven Projects. *IEEE Computer* 20, 7 (July), p. 91.

Foulser, D.E. and Schreiber, R. 1987. The Saxpy Matrix-1: A General-Purpose Systolic Computer. *IEEE Computer* 20, 7 (July), pp. 35-43.

FPS 1986. *The FPS T Series: A Parallel Vector Supercomputer* and *The FPS T Series*. Floating Point Systems, Inc.

FPS 1987. Software Brief, T Series Programming: Fortran-77 and C, Floating Point Systems, Inc.

Frenkel, K.A. 1986. Evaluating Two Massively Parallel Machines. *Communications of the ACM*

29, 8 (August), pp. 752-758.

Gajski, D.D., Padua, D.A., Kuck, D.J. and Kuhn, R.H. 1984. A Second Opinion on Data Flow Machines and Languages. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 489-500.

Ghosal, D. and Bhuyan, L.N. 1987. Performance Analysis of the MIT Tagged Token Dataflow Architecture. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 680-683.

Gilmore, P.A. 1986. The Massively Parallel Processor (MPP). *Supercomputers: Class VI Systems, Hardware and Software*, S. Fernbach, editor, Elsevier Science Publishers B.V. (North-Holland).

Goodyear. 1984. Functional Description of ASPRO, the High Speed Associative Processor. Goodyear Aerospace Corporation report 25500, (July 16).

Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L. and Snir, M. 1984. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 276-290.

Gottlieb, A. and Schwartz, J.T. 1982. Networks and Algorithms for Very-Large-Scale Parallel Computation. *IEEE Computer* 15, 1 (January), pp. 27-36.

Halstead, R.H., Jr. 1987. Overview of Concert Multilisp: A Multiprocessor Symbolic Computing System. *Computer Architecture* 15, 1 (March), pp. 5-14.

Hansen, P.B. 1975. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering* SE-1, 2 (June), pp. 313-321.

Harms, U. and Ronsch, W. 1986. International Conference on Vector and Parallel Computing. *Parallel Computing* 3, pp. 359-363.

Hayes, J.P., Mudge, T., Stout, Q.F., Colley, S. and Palmer, J. 1986. A Microprocessor-based Hypercube Supercomputer. *IEEE Micro*, (October), pp. 6-17.

Haynes, L.S. 1982. Highly Parallel Computing. *IEEE Computer* 15, 1, (January), pp. 7-8.

Haynes, L.S., Lau, R.L., Siewiorek, D.P. and Mizell, D.W. 1982. A Survey of Highly Parallel Computing. *IEEE Computer* 15, 1 (January), pp. 9-24.

Hays, N. 1986. New Systems Offer Near-Supercomputer Performance. *IEEE Computer* 19, 3 (March), pp. 104-107.

Hein, C.E., Ziegger, R.M. and Urbano, J.A. 1987. The Design of a GaAs Systolic Array for an Adaptive Null Steering Beamforming Controller. *IEEE Computer* 20, 7 (July), pp. 92-93.

Hillis, W.D. 1987. The Connection Machine. *Scientific American* 256, pp. 108-115.

Hillis, W.D. and Steele, G.L., Jr. 1986. Data Parallel Algorithms. *Communications of the ACM* 29, 12 (December), pp. 1170-1183.

Hillyer, B.K., Shaw, D.E. and Nigam, A. 1986. NON-VON's Performance on Certain Database Benchmarks. *IEEE Transactions on Software Engineering* SE-12, 4 (April), pp. 577-583.

Hoare, C.A.R. 1978. Communicating Sequential Processes. *Communications of the ACM* 21, 8

(August), pp. 666-77.

Hobson, N. 1987. San Diego Supercomputer Center. *Supercomputing* (Fall), pp. 23-24.

Hough, A.A. and Cuny, J.E. 1987. Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 735-738.

Howe, C.D. and Moxon, B. 1987. How to Program Parallel Processors. *IEEE Spectrum* 24, 6 (September), pp. 36-41.

Hwang, K. 1987. Advanced Parallel Processing with Supercomputer Architectures. *ICPP '87 Tutorial on Supercomputing Technologies, Architectures and Algorithms*, (August 21), pp. 1-94.

Hwang, K. and Briggs, F.A. 1984. *Computer Architecture and Parallel Processing*. McGraw-Hill, Inc.

Hwang, K. and Tseng, P.S. 1985. An Efficient VLSI Multiprocessor for Signal/Image Processing. *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, (October 7-10), pp. 172-175.

Hyde, D. 1987. The Connection Machine: Report on a Trip to the Naval Research Laboratory. *Supercomputer* 22, (November), pp. 37-40.

Hypercube 1985. *Proceedings of the First Conference on Hypercube Multiprocessors* . Society of Industrial and Applied Mathematics, Knoxville, TN, (August 26-27).

Hypercube 1986. *Proceedings of the Second Conference on Hypercube Multiprocessors* . Society of Industrial and Applied Mathematics.

IBM 1986. *Designing and Writing FORTRAN Programs for Vector and Parallel Processing*. IBM Corporation, SC23-0337-00, File No. S370-25, (November).

IEEE Report 1985. Software for High Peformance Computers. *IEEE Subcommittee on Supercomputers of the Committee on Communications and Information Policy*, (December).

IJSA 1987. *The International Journal of Supercomputer Applications* 1, 1 (Spring), pp. 115-116.

Ina, H., Kamiya, S. and Mikami, J. 1985. Languages and Software Development Tools for Supercomputers. *Computer Physics Communications* 38, pp. 211-19.

Intel 1987. *iPSC System Product Summary* and *Sugarcube Concurrent Computing Workstation Product Brief*, Intel Scientific Computers, Beaverton, Oregon.

IP1 1987. *Introducing IP-1*, International Parallel Machines, Inc.

Jajodia, S., Liu, J., and Ng. P.A. 1983. A Scheme of Parallel Processing for MIMD Systems. *IEEE Transactions on Software Engineering* SE-9, 4 (July), pp. 436-445.

Jensen, J.C. 1980. *Basic Program Representation in the Texas Instruments Data Flow Test Bed Compiler*, unpublished memo, Texas Instruments, Inc., (January).

Jesshope, C.R. 1986a. Computational Physics and the Need for Parallelism. *Computer Physics Communications* 41, pp. 363-375.

Jesshope, C.R. 1986b. Support for the Rapid Processing of Large Data Structures in Occam.

Colloquium on 'Software Engineering for VLSI Parallel Processing', Digest No. 102 (October), pp. 8/1-3.

Jordan, H.F. 1986. Structuring Parallel Algorithms in an MIMD, Shared Memory Environment. *Parallel Computing* 3, pp. 93-110.

Jurasek, D., Richardson, W. and Wilde, D. 1986. A Multiprocessor Design in Custom VLSI. *VLSI Systems Design*, (June), pp. 26-30.

Kandle, D.A. 1987. A Systolic Signal Processor for Signal-Processing Applications. *IEEE Computer* 20, 7 (July), pp. 94-95.

Kaplan, I. 1987. The LDF 100: A Large Grain Dataflow Parallel Processor. *Computer Architecture News* 15, 3 (June), pp. 5-12.

Karp, A.H. 1987. Programming for Parallelism. *Computer* 20, 5 (May), pp. 43-57.

Kartashev, S.P. and Kartashev, S.I. 1986. Adaptable Software for Dynamic Architectures. *IEEE Computer* 19, 2 (February), pp. 61-77.

Keller, R.M., Jayaraman, B., Rose, D. and Lindstrom, G. 1980. *FGL Programmer's Guide.* Department of Computer Science AMPS Technical Memo 1, University of Utah, Salt Lake City, Utah, (July).

Kerola, T. and Schwetman, H. 1987. Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs. *Proceedings of the 1987 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* 15, 1 (May), pp. 163-174.

Koelbel, C., Mehrotra, P. and van Rosendale, J. 1987. Semi-Automatic Domain Decomposition in BLAZE. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 521-524.

Koren, I. and Peled, I. 1987. The Concept and Implementation of Data-Driven Processor Arrays. *IEEE Computer* 20, 7 (July), pp. 102-103.

Kosinski, G.P. 1987. Super Growth in Minisupercomputers. *Supercomputing*, (Fall), pp. 28-33.

Kozdrowicki, E.W. and Theis, D.J. 1984. Second Generation of Vector Supercomputers. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 9-21.

Kuck, D.J. and Stokes, R.A. 1984. The Burroughs Scientific Processor (BSP). *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 90-103.

Kuck, D.J., Davidson, E.S., Lawrie, D.H. and Sameh, A.H. 1986. Parallel Supercomputing Today and the Cedar Approach. *Science* 231, (February 28), pp. 967-974.

Kung, H.T. 1982. Why Systolic Arrays? *IEEE Computer* 15, 1 (January), pp. 37-46.

Kung, S.Y., Lo, S.C., Jean, S.N. and Hwang, J.N. 1987. Wavefront Array Processors - Concept to Implementation. *IEEE Computer* 20, 7 (July), pp. 18-33.

Lannutti, J.E. 1985. Supercomputer Computations Research Institute, Tallahassee, Florida. *IEEE Software* 2, 6 (November), page 66.

Law, J.W. and Miller, N.A. 1983. ASPRO Technical Introduction. Operations Research

Information Control, La Jolla, CA.

Leblanc, T.J. and Mellor-Crummey, J.M. 1987. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers* C-36, 4 (April), pp. 471-482.

Leeland, S.B. 1987. An Advanced DSP Systolic-Array Architecture. *IEEE Computer* 20, 7 (July), pp. 95-96.

Lelick, M. 1985. Center for Supercomputing Research and Development, University of Illinois. *IEEE Software* 2, 6 (November), pp. 58-59.

Lester, B.P. and Guthrie, G.R. 1987. A System for Investigating Parallel Algorithm and Architecture Interaction. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 667-670.

Levitan, S.P., Weems, C.C., Hanson, A.R. and Riseman, E.M. 1987. The UMass Image Understanding System. *Parallel Computer Vision*, L. Uhr, editor, Academic Press, Inc., New York, pp. 215-248.

Li, K. and Schwetman, H. 1985. Vector C: A Vector Processing Language. *Journal of Parallel and Distributed Computing* 2, pp. 132-169.

Lin, W., Chin, C. and Ho, C. 1987. Integrating Systolic Arrays into a Supersystem. *IEEE Computer* 20, 7 (July), pp. 100-101.

Lincoln, N.R. 1982. Technology and Design Trade-offs in the Creation of a Modern Supercomputer. *IEEE Transactions on Computers* C-31, 5 (May), pp. 349-362.

Lopresti, D. P. 1987. P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences. *IEEE Computer* 20, 7 (July), pp. 98-99.

Loral ASPRO. 1982. ASPRO: ASsociative PROcessor: real-time performance for COMMAND and CONTROL. Loral Systems Group, Akron, Ohio.

Loral MPP. 1983. Massively Parallel Processor, A New Supercomputer for the 1980's. Loral Defense Systems Division Akron, Akron, Ohio.

Louter-Nool, M. 1987. Basic Linear Algebra Subprograms (BLAS) on the CSC CYBER 205. *Parallel Computing* 4, pp. 143-165.

Maeng, S.R. and Cho, J.W. 1987. A Control and Data Flow Multiprocessor. *The Australian Computer Journal*, 18, 1 (February), pp. 26-32.

Maisel, M. 1987. Science at the San Diego Supercomputer Center. *The International Journal of Supercomputer Applications* 1, 2 (Summer), pp. 6-10.

Maples, C. 1985. Analyzing Software Performance in a Multiprocessor Environment. *IEEE Software* 2, 4 (July), pp. 50-63.

Martin, J.L. 1985a. Operating Systems and Environments for Large-Scale Parallel Processing. *IEEE Software* 2, 4 (July), pp. 4-5.

Martin, J.L. 1985b. International Parallel Processing Projects: A Software Perspective. *IEEE Software* 2, 4 (July), pp. 65-80.

McCanny, J.V. and McWhirter, J.G. Some Systolic Array Developments in the United Kingdom. *IEEE Computer* 20, 7 (July), pp. 51-63.

McClain, C. 1987. SCS Supercomputing: Power to the People. *Supercomputing*, (Fall), pp. 4-8.

Mehrotra, P. and van Rosendale, J. 1987. The BLAZE Language: A Parallel Language for Scientific Programming. *Parallel Computing* 5, pp. 339-361.

Metcalf, M. 1987. Fortran 8x - The Emerging Standard. *Computer Physics Communications* 45, pp. 259-268.

Meyers, W. 1986. Getting the Cycles out of a Supercomputer. *IEEE Computer* 19, 3 (March), pp. 89-92.

Miller, R. 1984. ADLSIMD: An Abstract Description Language for the SIMD Family. *Proceedings of the 1984 IEEE Southern Tier Technical Conference*, pp. 89-94.

Miller, R. 1985. Writing SIMD Algorithms. *Proceedings of the 1985 International Conference on Computer Design: VLSI in Computers.*

Miller, R. and Miller, S.E. 1987. Using Hypercube Multiprocessors to Determine Geometric Properties of Digitized Pictures. *Proceedings of the 1987 International Conference on Parallel Processing*, (August 17-21), pp. 638-640.

Miller, R. and Stout, Q. F. 1987. *Parallel Algorithms for Regular Architectures.* The MIT Press, Cambridge, MA.

Miura, K. and Uchida, K. 1984. FACOM Vector Processor System: VP-100/VP-200. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 59-73.

Modi, J.J. and Rollett, J.S. 1986. Some Problems of Exploiting a Pipeline Processor. *Parallel Computing* 3, pp. 263-265.

Moore, R., Nassi, I., O'Neil, J. and Siewiorek, D.P. 1986. The Encore Multimax: A Multiprocessor Computing Environment. Encore Computer Corporation Technical Report ETR 86-004, (August 1).

Mudge, T.N. and Winsor, D.C. 1987. Multiple Bus Architectures, *IEEE Computer* 20, 6 (June), pp. 42-48.

Nagl, M. 1985. An Incremental Programming Support Environment. *Computer Physics Communications* 38, Amsterdam: North Holland, pp. 245-276.

Nash, G.J., Pryztula, W.K. and Hansen, S. 1987. The Systolic/Cellular System for Signal Processing. *IEEE Computer* 20, 6 (June), pp. 96-97.

Navarro, J.J., Llaberia, J.M. and Valero, M. 1987. Partitioning: An Essential Step in Mapping Algorithms Into Systolic Array Processors. *IEEE Computer* 20, 7 (July), pp. 77-89.

NCSA 1987. *NCSA Overview*; *Introducing the NCSA*; *Scientific Visualization Program*; *Industrial Supercomputing Program*; *System Overview*; *The NCSA Cray X-MP/48 System* and *The Macintosh as Workstation.*

Ncube 1987. *NCUBE ten: An Overview - True Parallel Computing*, *The NCUBE seven*, and *The NCUBE four*, Ncube Corporation.

Norin, R.S. and Smith, W.R. 1986. Array Processors Speed Radar Signal Processing. *Computer Design*, (April 1), pp. 69-72.

Nunns, T. 1987. Supercomputing in Western Canada. *The International Journal of Supercomputer Applications* 1, 3 (Fall), pp. 5-11.

Olson, R. 1985. Parallel Processing in a Message-Based Operating System. *IEEE Software* 2, 4 (July), pp. 39-49.

Orszag, S.A. 1985. John von Neumann Center for Scientific Computing, Princeton, New Jersey. *IEEE Software* 2, 6 (November), pp. 63-64.

Padua, D.A., Kuck, D.J., and Lawrie, D.H. 1980. High-Speed Multiprocessors and Compilation Techniques. *IEEE Transactions on Computers* C-29, 9 (September), pp. 763-776.

Padua, D.A., Guarna, V.A. and Lawrie, D.H. 1987. *Supercomputer Programming Environments.* Center for Supercomputing Research and Development Report no. 673, University of Illinois at Urbana-Champaign, (June 9).

Padua, D.A. and Wolfe, M.J. 1986. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM* 29, 12 (December), pp. 1184-1201.

Palumbo, P. 1987. The MPP. Term Paper for Parallel Algorithms (CS633) under R. Miller, State University of New York, Buffalo, New York.

Parkinson, D. 1986. Serial to Parallel - A Revolution in Processing. *IEE Colloquium on 'Software Engineering for VLSI Parallel Processors'*, Digest No. 102 (October), p. 2/1.

Passero, E. 1981. A Multiprocessor Monitor Debugger. *International Electrical, Electronics Conference and Exposition*, pp. 72-73.

Paul, G. 1984. VECTRAN and the Proposed Vector/Array Extensions to ANSI FORTRAN for Scientific and Engineering Computation. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 143-162.

Perrott, R.H. 1979. A Language for Array and Vector Processors. *ACM Transactions on Programming Languages and Systems* 1, 2 (October), pp. 177-95.

Perrott, R.H., Crookes, D., Milligan, P. and Purdy, W. R. M. 1985. A Compiler for an Array and Vector Processing Language. *IEEE Transactions on Software Engineering* SE-11, 5 (May), pp. 471-478.

Perrott, R.H., Lyttle, R.W. and Dhillon, P.S. 1987. The Design and Implementation of a Pascal-Based Language for Array Processor Architectures. *Journal of Parallel and Distributed Computing* 4, 3 (June), pp. 266-287.

Perrott, R.H. and Zarea-Aliabadi, A. 1986. Supercomputer Languages. *Computing Surveys* 18, 1 (March), pp. 5-22.

Perry, T. 1987. NASA Unveils Computer for Aerodynamics Research. IEEE's *The Institute* 11, 8 (August), p. 4.

Pfeiffer, W. 1985. Computing at the San Diego Supercomputer Center, University of California. *IEEE Software* 2, 6 (November), pp. 60-62.

Pfister, G.F., Brantley, W.C., George, D.A., Harvey, S.L., Kleinfelder, W.J., McAuliffe, K.P., Melton, E.A., Norton, V.A. and Weiss, J. 1986. An Introduction to the IBM Research Parallel Processor Prototype (RP3). Research Report #53834, IBM T.J. Watson Research Center, Yorktown Heights, New York, (June 13).

Potter, J.L., editor. 1985. *The Massively Parallel Processor*. The MIT Press, Cambridge, Massachusetts.

Pratt, T.W. 1985. Pisces: An Environment for Parallel Scientific Computation. *IEEE Software* 2, 4 (July), pp. 7-20.

Pratt, T.W. 1987. The Pisces 2 Parallel Programming Environment. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 439-445.

Prywes, N., Shi, Y., Szymanski, B. and Tseng, J. 1986. Supersystem Programming with Model. *IEEE Computer* 19, 2 (February), pp. 50-59.

PSC 1987. *Projects in Scientific Computing* and an information sheet, Pittsburgh Supercomputer Center.

Purtilo, J., Reed, D.A. and Grunwald, D.C. 1987. Environments for Prototyping Parallel Algorithms. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 431-438.

Quinn, M.J. 1987. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, Inc., New York.

Raghavan, R. 1986. Processing on Geometric Single-Instruction Multiple-Data Machines. *Colloquium on 'Software Engineering for VLSI Parallel Processing'*, Digest No. 102, (October), pp. 5/1-3.

Ramakrishnan, I.V. and Browne, J.C. 1983. A Paradigm for the Design of Parallel Algorithms with Applications. *IEEE Transactions on Software Engineering* SE-9, 4 (July), pp. 411-415.

Reed D.A. and Grunwald, D.C. 1987. The Performance of Multicomputer Interconnection Networks. *IEEE Computer* 20, 6 (June), pp. 63-73.

Reeves, A.P. 1984. Parallel Pascal: An Extended Pascal for Parallel Computers. *Journal of Parallel and Distributed Computing* 1, pp. 64-80.

Reeves, A.P. and Bergmark, D. 1987. Parallel Pascal and the FPS Hypercube Supercomputer. *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 385-388.

Rettberg, R. and Thomas, R. 1986. Contention is no Obstacle to Shared-Memory Multiprocessing. *Communications of the ACM* 29, 12 (December), pp. 1202-1212.

Rice, J.R. 1987. Barriers to the use of Supercomputers. *SIAM News* 20, 5 (September), pp. 13, 17.

Rodrigue, G., Giroux, E.D. and Pratt, M. 1984. Large-Scale Scientific Computation. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 429-451.

Rogers, J.L., Jr. and Sobieszczanski-Sobieski, J. 1987. Exploiting Parallel Computing with Limited Program Changes Using a Network of Microcomputers. *Advances in Engineering Software* 9, 1, pp. 29-33.

Russell, C.H. and Waterman, P.J. 1987. Variations on Unix for Parallel-Processing Computers. *Communications of the ACM* 30, 12 (December), pp. 1048-1055.

Schneck, P. 1985. Supercomputing Research Center, Greenbelt, Maryland. *IEEE Software* 2, 6

(November), p. 59.

Schoitsch, E. 1986. Software Engineering Aspects of Real-Time Programming Concepts. *Computer Physics Communications* 41, pp. 327-361.

Schwan, K. and Jones, A.K. 1986. Flexible Software Development for Multiple Computer Systems. *IEEE Transactions on Software Engineering* SE-12, 3 (March), pp. 385-401.

Schwederski, T. and Siegel, H.J. 1986. Adaptable Software for Supercomputers. *IEEE Computer*, (February), pp. 40-48.

Schwetman, H.D. 1986. PPL: A Parallel Programming Language based on C. *Microelectronics and Computer Technology Corporation Technical Report, PP-096-86*, (March).

SCRI 1985. Supercomputer Computations Research Institute pamphlet.

SDSC 1987. *An Overview of the San Diego Supercomputer Center*, San Diego Supercomputer Center, (April 30).

Segall, Z. and Rudolph, L. 1985. PIE: A Programming and Instrumentation Environment for Parallel Processing, *IEEE Software* 2, 4 (July), pp. 22-37.

Sequent 1986. Balance Technical Summary, MAN-0110-00, (November 19), Sequent Computer Systems, Incorporated, Beaverton, Oregon.

Sequent 1987. Sequent B8 and B21 Parallel Computer Systems and Sequent S27 and S81 Parallel Computer Systems, Sequent Computer Systems, Incorporated, Beaverton, Oregon.

Sherson, I.D. and Ma, Y. 1987. Orthogonal Access Multiprocessing: An Architecture for Numerical Applications. *Proceeding of the 8th Symposium on Computer Arithmetic*, Lake Como, Italy, (May 18-21).

SIAM 1987. *SIAM News* 20, 5 (September).

Siegel, H.J. 1979. Interconnection Networks for SIMD Machines. *IEEE Computer* 12, 6 (June), pp. 57-65.

Siegel, H.J. 1984. *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*. Lexington Books, Lexington, MA.

Siegel, H.J., Hsu, T.W. and Jeng, M. 1987. An Introduction to the Multistage Cube Family of Interconnection Networks. *The Journal of Supercomputing* 1, 1, pp. 13-42.

Siegel, H.J., Schwederski, T., Kuehn, J.T. and Davis IV, N.J. 1987. An Overview of the PASM Parallel Processing System. *Tutorial: Computer Architecture*, edited by Gajski, D. D., Milutinovic, V. M. Siegel, H. J. and Furht, B. P. IEEE Computer Society Press, Washington, D.C., pp. 387-407.

Smith, B.J. 1984. Architecture and Applications of the HEP Multiprocessor Computer System. *Tutorial Supercomputers: Design and Applications*, edited by Hwang, K., IEEE Computer Society Press, pp. 231-238.

So, K., Bolmarcich, A.S., Darema, F. and Norton, V.A. 1987. A Speedup Analyzer for Parallel Programs. *Proceedings of the 1987 International Conference on Parallel Processing*, (August), pp. 653-662.

Soll, D.B. 1986. *Vectorization and Vector Migration Techniques*. IBM Technical Bulletin, R.

Stephens, editor, (June).

Sonnenschein, M. An Fxtension of the Language C for Concurrent Programming. *Parallel Computing* 3, pp. 59-71.

SRC 1986a. *Report of the Summer Workshop on Parallel Algorithms and Architectures.* Supercomputing Research Center, Lanham, MD, (February).

SRC 1986b. *Second Workshop on Supercomputing. Research and Technology Issues in Supercomputing.* Supercomputing Research Center, Lanham, MD, (December).

Srini, V.P. 1986. An Architectural Comparison of Dataflow Systems. *IEEE Computer* 19, 3 (March), pp. 68-88.

Stafford, J. 1987. CELERITY Describes System for Supercomputing at the Departmental Level. *Supercomputing*, (Fall), pp. 36-39.

Stanfill, C. and Kahle, B. 1986. Parallel Free-text Search on the Connection Machine System. *Communications of the ACM* 29, 12 (December), pp. 1229-1239.

Stolfo, S. and Miranker, D.P. 1984. DADO: A Parallel Processor for Expert Systems. *Proceedings of the 1984 International Conference on Parallel Processing*, pp. 74-82.

Supercomputer 1987. *The International Journal of Supercomputer Applications* 1, 1 (Spring), p. 116.

Supercomputing 1987a. Advertisement. *SuperComputing* (Fall), pp. 20-21.

Supercomputing 1987b. Advertisement. *SuperComputing* (Fall), page 22.

Szymanski, B.K. and Mueller-Wichards, D. 1987. Parallel Programming with Recurrent Equations. *The International Journal of Supercomputer Applications* 1, 2 (Summer), pp. 44-74.

Tanimoto, S.L. 1982. Advances in Software Engineering and their Relations to Pattern Recognition and Image Processing. *Pattern Recognition* 15, 3, pp. 113-120.

Taylor, S., Hellerstein, L., Safra, S. and Shapiro, E. 1987. Notes on the Complexity of Systolic Programs. *Journal of Parallel and Distributed Computing* 4, pp. 250-265.

TMC 1987. Connection Machine Model CM-2 Technical Summary (April) and The Connection Machine Family, Thinking Machines Corporation.

Trakhtengerts, E.A. and Shuraits, Y. 1982. Software Design for Multiprocessor Systems Computer Control. *Proceedings of the 4th IFAC Workshop*, pp. 1-12.

Tucker, N.D. 1986. Development and Application of Parallel Processing. *Data Processing* (GB) 28, 8 (October), pp. 405-409.

Tuomenoksa, D.L. and Siegel, H.J. 1985. Task Scheduling on the PASM Parallel Processing System. *IEEE Transactions on Software Engineering* SE-11, 2 (February), pp. 145-157.

Tutorial 1981. Introduction, pp. 1-10; Pipeline Processors, p. 25; Multiprocessors, p. 135; Dataflow Processors, p. 209; Special-Purpose Parallel Architectures, p. 235; Parallel Programming Languages, p. 293 and Parallelism Extraction. *Tutorial on Parallel Processing*, p. 345.

Tutorial 1984. Introduction, pp. 2-3; Modern Supercomputers, pp. 5-8; Vector Supercomputers, pp. 24-30; Multiprocessing Supercomputers, pp. 206-212; Algorithms and Applications, pp. 358-359; Future Trends in Supercomputers, pp. 476-477. *Tutorial Supercomputers: Design and Applications*. Hwang, K., editor. IEEE Computer Society Press.

Uhr, L. 1984. *Algorithm-Structured Computer Arrays and Networks*. Academic Press, New York.

Unix 1984. *UNIX User's Manual, Reference Guide - 4.2 Berkeley Software Distribution*, Computer Science Division, University of California, Berkeley, CA.

Wallich, P. 1987. Thinking Machines Corp. adds Megaflops to MIPS. *The Institute* 11, 7 (July).

Wallqvist, A. and Berne, B.J. 1987. Two Examples from Chemical Physics. *IEEE Computer* 20, 5 (May), pp. 9-21.

Wayman, R. 1986. Software Engineering for Transputer-Based Systems. *Colloquium on 'Software Engineering for VLSI Parallel Processing'*, Digest No. 102, (October), pp. 7/1-3.

Wegner, P. and Smolka, S.A. 1983. Processes, Tasks, and Monitors: A Comparative Study of Concurrent Programming Primitives, *IEEE Transactions on Software Engineering* SE-9, 4 (July), pp. 446-62.

Welch, H.O. 1984. Software Development for Array Machines. *Handbook of Software Engineering*, edited by Vick, C. R. and Ramamoorthy, C. V., Van Nostrand Reinhold Company, NY, pp. 623-639.

Whitby-Strevens, C. 1985. The Transputer. *Proceedings of the Symposium on Computer Architectures*, pp. 292-300.

Wilhelmson, R.B. 1985. National Center for Supercomputing Applications, University of Illinois. *IEEE Software* 2, 6 (November), pp. 65-66.

Xu, Z. and Hwang, K. 1987. Molecules: A Language Construct for Concurrent Programming. *IEEE Transactions on Software Engineering*, to appear.